Universidade Federal de Uberlandia
Faculdade de Computação

Construção de um Compilador de JavaScript para Webassembly usando Ocaml

Aluno: Amanda Silva Moreira
Matricula: 11611BCC042
Professor: Alexsandro Soares

Junho
2019

Sumário

1 Introdução

Este trabalho aborda o processo de construção de um compilador para uma mini linguagem. Para iniciar, este capítulo abordará os procedimentos de instalação do OCaml e da plataforma WebAssembly, ferramentas utilizadas na contrução do compilador de JavaScript para WebAssembly

1.1 OCaml

Objective Caml, ou OCaml(Objective Categorical Abstract Machine Lan- guage) é uma linguagem de programação funcional fortemente tipada, com tipagem estática e inferência de tipos, também admite a programação imperativa e orientada a objetos. Nesse trabalho, a linguagem OCaml será utilizada para a implementação do compilador.

1.2 WebAssembly

WebAssembly (Wasm abreviado) é um formato de instrução binária para máquinas virtuais baseada em pilha. O Wasm é projetado como um alternativa portátil para a compilação de linguagens de alto nível como C / C ++ /Rust, permitindo a implantação na web para aplicativos clientes e servidores.

1.3 JavaScript

JavaScript é uma linguagem de programação interpretada de alto nível, caracterizada também como dinâmica, fracamente tipada, prototype-based e multi-paradigma. Juntamente com HTML e CSS, o JavaScript é uma das três principais tecnologias da World Wide Web.

2 Configurando Ambiente

2.1 Ubuntu

A versão 18.04.2 LTS da distribuição Ubuntu do sistema operacional Linux está sendo usado para a realização deste trabalho. O download pode ser realizado pelo seguinte link: https://www.ubuntu.com/download/desktop.
O Ubuntu foi instalado em dual boot e nesse processo houve problemas com a BIOS do notebook, foi necessário a atualização da mesma. O pacote de atualização da BIOS pode ser encontrado no site da marca do computador/notebook, basta buscar pelo modelo do mesmo.

2.2 Ocaml e bibliotecas

É recomendado a instalação do opam, gestor de pacotes de software, para a instalação de ferramentas e bibliotecas. O OCaml é instalado pelo opam.

```
sudo apt install wget
wget https://raw.github.com/ocaml/opam/master/shell/opam_installer.sh -O
    - | sh -s /usr/local/bin
opam init
eval `opam config env`
opam repository add git git+https://github.com/ocaml/opam-repository
opam update
```

Para verificar a versão do opam:

```
opam --version
$ Versão instalada 1.2.2 $
```

Para verificar a versão do OCaml:

```
ocaml -version
$ Versão instalada 4.05.0 $
```

É necessário instalar o rlwrap, editor do OCaml:

```
sudo apt install rlwrap
```

2.3 JavaScript

É necessário a instalação do Node[1] para executar códigos JavaScript.

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.34.0/
   install.sh | bash
node -v
$ Versão instalada v8.9.1 $
```

A melhor forma de instalação do Node encontrada é pelo Git.

2.4 WebAssembly

Como não foi encontrado uma boa forma de converter de JavaScript para WebAssembly, será feita a converção de C para WebAssembly.
Uma ferramenta pré-compilada para converter de C para o WebAssembly é facilmente obtido através do GitHub[2] instalando o emscripten.

```
git clone https://github.com/emscripten-core/emsdk.git
cd emsdk
./emsdk install latest
./emsdk activate latest
source ./emsdk_env.sh
cd ..
```

2.4.1 Compile e execute

Agora, com as ferramentas necessárias completas, que podemos usar para compilar um programa para o WebAssembly. Entao faz o mesmo processo para todos os programas nano, exemplo:

```
emcc example.c -s EXIT_RUNTIME=1 -o example.js
node example.js
```

Assim o programa é executado e a sua resposta aparece no terminal.

2.4.2 Curiosidade

Os arquivos também podem ser vistos via HTTP usando um servidor web emrum fornecido com o Emscripten SDK:

```
emcc example.c -s WASM=1 -o example.html
emrun --no_browser --port 8080 .
```

2.4.3 Gerando o código .wat

O código gerado pelo emscripten é .wasm e a visualização do código assembly é dificultada, por isso convertemos para .wat com o WABT[2]: O kit de ferramentas binárias do WebAssembly.
O WABT (nós o pronunciamos "wabbit") é um conjunto de ferramentas para o WebAssembly, que são destinadas ao uso em(ou para desenvolvimento de) toolchains ou outros sistemas que desejem manipular arquivos WebAssembly.
Passos para uso do WABT:

```
git clone --recursive https://github.com/WebAssembly/wabt
cd wabt
mkdir build
cd build
sudo apt install cmake
cmake ..
cd wabt
make gcc-release
sudo make gcc-release
cd out/gcc/Release
cd wabt
export PATH=`pwd`/out/gcc/Release
```

Agora, na mesma pasta dos arquivos wasm que serão convertidos em wat:

```
wasm2wat nano01.wasm > nano01.wat
wasm2wat nano02.wasm > nano02.wat
wasm2wat nano03.wasm > nano03.wat
wasm2wat nano04.wasm > nano04.wat
wasm2wat nano05.wasm > nano05.wat
```

```
wasm2wat nano06.wasm > nano06.wat
wasm2wat nano07.wasm > nano07.wat
wasm2wat nano08.wasm > nano08.wat
wasm2wat nano09.wasm > nano09.wat
wasm2wat nano10.wasm > nano10.wat
wasm2wat nano11.wasm > nano11.wat
wasm2wat nano12.wasm > nano12.wat
```

Este passo gera os códigos .wat, mas existe um problema, o código gerado é extenso como no exemplo a seguir:
nano01.wat

```
(module
  (type (;0;) (func (param i32 i32 i32) (result i32)))
  (type (;1;) (func (param i32) (result i32)))
  (type (;2;) (func (param i32)))
  (type (;3;) (func (param i32 i32) (result i32)))
  (type (;4;) (func (result i32)))
  (type (;5;) (func (param i32 i32)))
  (type (;6;) (func))
  (type (;7;) (func (param i32 i32 i32 i32) (result i32)))
  (import "env" "abortStackOverflow" (func (;0;) (type 2)))
  (import "env" "nullFunc_ii" (func (;1;) (type 2)))
  (import "env" "nullFunc_iiii" (func (;2;) (type 2)))
  (import "env" "___lock" (func (;3;) (type 2)))
  (import "env" "___setErrNo" (func (;4;) (type 2)))
  (import "env" "___syscall140" (func (;5;) (type 3)))
  (import "env" "___syscall146" (func (;6;) (type 3)))
  (import "env" "___syscall54" (func (;7;) (type 3)))
  (import "env" "___syscall6" (func (;8;) (type 3)))
  (import "env" "___unlock" (func (;9;) (type 2)))
  (import "env" "_emscripten_get_heap_size" (func (;10;) (type 4)))
  ...
```

Esse código tem gerado tem 9383 linhas, por isso usei a plataforma WABT[4] online, que converte de .c para .wat e gera um código assembly menor, como mostrado na próxima sessão.

2.4.4 Observações

Um problema encontrado, após o uso do wabt é que alguns comandos ficam desabilitados e para poder usa-los é necessário usar o seguinte comando no terminal:

```
export PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
    :/usr/games"
```

3 Códigos

3.1 JavaScript

nano01

```
function main(): void = {
}

main();
```

nano02

```
function main(): void = {
    var n: integer;
}

main();
```

nano03

```
function main(): void = {
    var n: integer;

    n = 1;
}

main();
```

nano04

```
function main(): void = {
    var n: integer;

    n = 1 + 2;
}

main();
```

nano05

```
function main(): void = {
    var n: integer;

    n = 2;
    console.log(n);
}

main();
```

namo06

```
function main(): void = {
    var n: integer;

    n = 1 - 2;
    console.log(n);
}

main();
```

nano07

```
function main(): void = {
    var n: integer;

    n = 1;
    if (n === 1) {
        console.log(n);
    }
}

main();
```

nano08

```
function main(): void = {
    var n: integer;

    n = 1;
    if (n === 1) {
        console.log(n);
    } else {
        console.log(0);
    }
}

main();
```

nano09

```
function main(): void = {
    var n: integer;

    n = 1 + 1 / 2;
    if (n === 1) {
        console.log(n);
    } else {
        console.log(0);
    }
}

main();
```

nano10

```
function main(): void = {
    var n: integer;
    var m: integer;

    n = 1;
    m = 2;
    if (n === m) {
        console.log(n);
    } else {
        console.log(0);
    }
}

main();
```

nano11

```
function main(): void = {
    var n: integer;
    var m: integer;
    var x: integer;

    n = 1;
    m = 2;
    x = 5;
    while (x > n) {
        n = n + m;
        console.log(n);
    }
}

main();
```

nano12

```
function main(): void = {
    var n: integer;
    var m: integer;
    var x: integer;

    n = 1;
    m = 2;
    x = 5;
    while (x > n) {
        if (n === m) {
            console.log(n);
        } else {
            console.log(0);
        }
    }
}

main();
```

3.2 C

nano01

```
void main() {
}
```

nano02

```
void main() {
  int n;
}
```

nano03

```
void main() {
  int n;
  n = 1;
}
```

nano04

```
void main() {
  int n;
  n = 1 + 2;
}
```

nano05

```
#include <stdio.h>

void main() {
  int n;
  n = 2;
  printf("%d", n);
}
```

nano06

```
#include <stdio.h>

void main() {
  int n;
  n = 1 - 2;
  printf("%d", n);
}
```

nano07

```
#include <stdio.h>

void main() {
  int n;
  n = 1;
  if (n == 1)
    printf("%d", n);
}
```

nano08

```
#include <stdio.h>

void main() {
  int n;

  n = 1;
  if (n == 1)
    printf("%d", n);
  else
    printf("%d", "0");
}
```

nano09

```
#include <stdio.h>

void main() {
  int n;

  n = 1 + 1 / 2;
  if (n == 1)
    printf("%d", n);
  else
```

```
        printf("%d", 0);
    }
```

nano10

```
    #include <stdio.h>

    void main() {
      int n, m;
      n = 1;
      m = 2;

     if ( n == m)
        printf("%d", n);
     else
        printf("%d", 0);

    }
```

nano11

```
    #include <stdio.h>

    void main() {
      int n, m, x;
      n = 1;
      m = 2;
      x = 5;

      while (x > n) {
        n = n + m;
        printf("%d", n);
      }
    }
```

nano12

```
    #include <stdio.h>

    void main() {
      int n, m, x;
      n = 1;
      m = 2;
      x = 5;

      while (x > n) {
        if (n == m)
          printf("%d", n);
        else
          printf("%d", "0");
        x = x - 1;
      }
    }
```

3.3 WebAssembly

nano01

```
(module
 (table 0 anyfunc)
```

```
  (memory $0 1)
  (export "memory" (memory $0))
  (export "main" (func $main))
  (func $main (; 0 ;) (result i32)
   (i32.const 0)
  )
 )
```

nano02

```
 (module
  (table 0 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "main" (func $main))
  (func $main (; 0 ;) (result i32)
   (i32.const 0)
  )
 )
```

nano03

```
 (module
  (table 0 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "main" (func $main))
  (func $main (; 0 ;) (result i32)
   (i32.const 0)
  )
 )
```

nano04

```
 (module
  (table 0 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "main" (func $main))
  (func $main (; 0 ;) (result i32)
   (i32.const 0)
  )
 )
```

nano05

```
 (module
  (type $FUNCSIG$ii (func (param i32) (result i32)))
  (type $FUNCSIG$iii (func (param i32 i32) (result i32)))
  (import "env" "printf" (func $printf (param i32 i32) (result i32)))
  (table 0 anyfunc)
  (memory $0 1)
  (data (i32.const 16) "%d\n\00")
  (export "memory" (memory $0))
  (export "main" (func $main))
  (func $main (; 1 ;) (result i32)
   (local $0 i32)
   (i32.store offset=4
    (i32.const 0)
    (tee_local $0
     (i32.sub
```

```
      (i32.load offset=4
       (i32.const 0)
      )
      (i32.const 16)
     )
    )
   )
   (i32.store
    (get_local $0)
    (i32.const 2)
   )
   (drop
    (call $printf
     (i32.const 16)
     (get_local $0)
    )
   )
   (i32.store offset=4
    (i32.const 0)
    (i32.add
     (get_local $0)
     (i32.const 16)
    )
   )
   (i32.const 0)
  )
 )
```

nano06

```
(module
(type $FUNCSIG$ii (func (param i32) (result i32)))
(type $FUNCSIG$iii (func (param i32 i32) (result i32)))
(import "env" "printf" (func $printf (param i32 i32) (result i32)))
(table 0 anyfunc)
(memory $0 1)
(data (i32.const 16) "%d\n\00")
(export "memory" (memory $0))
(export "main" (func $main))
(func $main (; 1 ;) (result i32)
 (local $0 i32)
 (i32.store offset=4
  (i32.const 0)
  (tee_local $0
   (i32.sub
    (i32.load offset=4
     (i32.const 0)
    )
    (i32.const 16)
   )
  )
 )
 (i32.store
  (get_local $0)
  (i32.const -1)
 )
 (drop
  (call $printf
   (i32.const 16)
   (get_local $0)
```

```
    )
   )
   (i32.store offset=4
    (i32.const 0)
    (i32.add
     (get_local $0)
     (i32.const 16)
    )
   )
   (i32.const 0)
  )
 )
```

nano07

```
 (module
  (type $FUNCSIG$ii (func (param i32) (result i32)))
  (type $FUNCSIG$iii (func (param i32 i32) (result i32)))
  (import "env" "printf" (func $printf (param i32 i32) (result i32)))
  (table 0 anyfunc)
  (memory $0 1)
  (data (i32.const 16) "%d\n\00")
  (export "memory" (memory $0))
  (export "main" (func $main))
  (func $main (; 1 ;) (result i32)
   (local $0 i32)
   (i32.store offset=4
    (i32.const 0)
    (tee_local $0
     (i32.sub
      (i32.load offset=4
       (i32.const 0)
      )
      (i32.const 16)
     )
    )
   )
   (i32.store
    (get_local $0)
    (i32.const 1)
   )
   (drop
    (call $printf
     (i32.const 16)
     (get_local $0)
    )
   )
   (i32.store offset=4
    (i32.const 0)
    (i32.add
     (get_local $0)
     (i32.const 16)
    )
   )
   (i32.const 0)
  )
 )
```

nano08

```
(module
 (type $FUNCSIG$ii (func (param i32) (result i32)))
 (type $FUNCSIG$iii (func (param i32 i32) (result i32)))
 (import "env" "printf" (func $printf (param i32 i32) (result i32)))
 (table 0 anyfunc)
 (memory $0 1)
 (data (i32.const 16) "%d\n\00")
 (export "memory" (memory $0))
 (export "main" (func $main))
 (func $main (; 1 ;) (result i32)
  (local $0 i32)
  (i32.store offset=4
   (i32.const 0)
   (tee_local $0
    (i32.sub
     (i32.load offset=4
      (i32.const 0)
     )
     (i32.const 16)
    )
   )
  )
  (i32.store
   (get_local $0)
   (i32.const 1)
  )
  (drop
   (call $printf
    (i32.const 16)
    (get_local $0)
   )
  )
  (i32.store offset=4
   (i32.const 0)
   (i32.add
    (get_local $0)
    (i32.const 16)
   )
  )
  (i32.const 0)
 )
)
```

nano09

```
(module
 (type $FUNCSIG$ii (func (param i32) (result i32)))
 (type $FUNCSIG$iii (func (param i32 i32) (result i32)))
 (import "env" "printf" (func $printf (param i32 i32) (result i32)))
 (table 0 anyfunc)
 (memory $0 1)
 (data (i32.const 16) "%d\n\00")
 (export "memory" (memory $0))
 (export "main" (func $main))
 (func $main (; 1 ;) (result i32)
  (local $0 i32)
  (i32.store offset=4
   (i32.const 0)
   (tee_local $0
    (i32.sub
```

```
      (i32.load offset=4
       (i32.const 0)
      )
      (i32.const 16)
     )
    )
   )
   (i32.store
    (get_local $0)
    (i32.const 1)
   )
   (drop
    (call $printf
     (i32.const 16)
     (get_local $0)
    )
   )
   (i32.store offset=4
    (i32.const 0)
    (i32.add
     (get_local $0)
     (i32.const 16)
    )
   )
   (i32.const 0)
  )
 )
```

nano10

```
 (module
  (type $FUNCSIG$ii (func (param i32) (result i32)))
  (type $FUNCSIG$iii (func (param i32 i32) (result i32)))
  (import "env" "printf" (func $printf (param i32 i32) (result i32)))
  (table 0 anyfunc)
  (memory $0 1)
  (data (i32.const 16) "%d\n\00")
  (export "memory" (memory $0))
  (export "main" (func $main))
  (func $main (; 1 ;) (result i32)
   (local $0 i32)
   (i32.store offset=4
    (i32.const 0)
    (tee_local $0
     (i32.sub
      (i32.load offset=4
       (i32.const 0)
      )
      (i32.const 16)
     )
    )
   )
   (i32.store
    (get_local $0)
    (i32.const 0)
   )
   (drop
    (call $printf
     (i32.const 16)
     (get_local $0)
```

```
    )
   )
   (i32.store offset=4
    (i32.const 0)
    (i32.add
     (get_local $0)
     (i32.const 16)
    )
   )
   (i32.const 0)
  )
 )
```

nano11

```
(module
 (type $FUNCSIG$ii (func (param i32) (result i32)))
 (type $FUNCSIG$iii (func (param i32 i32) (result i32)))
 (import "env" "printf" (func $printf (param i32 i32) (result i32)))
 (table 0 anyfunc)
 (memory $0 1)
 (data (i32.const 16) "%d\00")
 (export "memory" (memory $0))
 (export "main" (func $main))
 (func $main (; 1 ;) (result i32)
  (local $0 i32)
  (local $1 i32)
  (i32.store offset=4
   (i32.const 0)
   (tee_local $1
    (i32.sub
     (i32.load offset=4
      (i32.const 0)
     )
     (i32.const 16)
    )
   )
  )
  (set_local $0
   (i32.const -1)
  )
  (loop $label$0
   (i32.store
    (get_local $1)
    (i32.add
     (get_local $0)
     (i32.const 4)
    )
   )
   (drop
    (call $printf
     (i32.const 16)
     (get_local $1)
    )
   )
   (br_if $label$0
    (i32.lt_s
     (tee_local $0
      (i32.add
       (get_local $0)
```

```
        (i32.const 2)
       )
      )
      (i32.const 3)
     )
    )
   )
   (i32.store offset=4
    (i32.const 0)
    (i32.add
     (get_local $1)
     (i32.const 16)
    )
   )
   (i32.const 0)
  )
 )
```

nano12

```
(module
 (type $FUNCSIG$ii (func (param i32) (result i32)))
 (type $FUNCSIG$iii (func (param i32 i32) (result i32)))
 (import "env" "printf" (func $printf (param i32 i32) (result i32)))
 (table 0 anyfunc)
 (memory $0 1)
 (data (i32.const 16) "%d\n\00")
 (export "memory" (memory $0))
 (export "main" (func $main))
 (func $main (; 1 ;) (result i32)
  (local $0 i32)
  (local $1 i32)
  (i32.store offset=4
   (i32.const 0)
   (tee_local $1
    (i32.sub
     (i32.load offset=4
      (i32.const 0)
     )
     (i32.const 16)
    )
   )
  )
  (set_local $0
   (i32.const 6)
  )
  (loop $label$0
   (i32.store
    (get_local $1)
    (i32.const 0)
   )
   (drop
    (call $printf
     (i32.const 16)
     (get_local $1)
    )
   )
   (br_if $label$0
    (i32.gt_s
     (tee_local $0
```

```
        (i32.add
         (get_local $0)
         (i32.const -1)
        )
       )
       (i32.const 2)
      )
     )
    )
    (i32.store offset=4
     (i32.const 0)
     (i32.add
      (get_local $1)
      (i32.const 16)
     )
    )
    (i32.const 0)
   )
  )
```

4 Analisador Léxico

O analisador léxico recebe o código fonte e a partir de uma sequência de caracteres produz uma sequência de símbolos chamados tokens. Também elimina espaços em branco e comentários, além de identificar erros léxicos, simplificando a etapa da análise sintática.
Para isso será usado o gerador de analisadores lexicais da linguagem OCaml, o Ocamellex. Este gerador produz o código do analisador léxico a partir de uma especificação lexial.
A especificação deve conter a definição dos tokens da linguagem (representados por expressões regulares) e a ação tomada para cada token.
O código em Ocaml a seguir apresenta uma especificação lexical simplificada, que deve ser salvo como um arquivo '.mll'.

lexico.mll

```
{
  open Lexing
  open Printf

  let incr_num_linha lexbuf =
    let pos = lexbuf.lex_curr_p in
     lexbuf.lex_curr_p <- { pos with
        pos_lnum = pos.pos_lnum + 1;
        pos_bol = pos.pos_cnum;
     }

  let msg_erro lexbuf c =
    let pos = lexbuf.lex_curr_p in
    let lin = pos.pos_lnum
    and col = pos.pos_cnum - pos.pos_bol - 1 in
    sprintf "%d-%d: caracter desconhecido %c" lin col c

  let erro lin col msg =
    let mensagem = sprintf "%d-%d: %s" lin col msg in
       failwith mensagem
```

```
type tokens = APAR
            | FPAR
            | ACHAVE
            | FCHAVE
            | ACOLCH
            | FCOLCH
            | MAIS
            | MENOS
            | MULT
            | DIV
            | MOD
            | POT
            | MAIOR
            | MENOR
            | ATRIB
            | IGUAL
            | MAISATRIB
            | MENOSATRIB
            | DIVATRIB
            | MULTATRIB
            | MENORIGUAL
            | MAIORIGUAL
            | INCR
            | DECR
            | AND
            | OR
            | NOT
            | VIRG
            | PONTOVIRG
            | DIF
            | FOR
            | IF
            | ELSE
            | WHILE
            | SWITCH
            | CASE
            | BREAK
            | DEFAULT
            | CONSOLELOG
            | FUNCTION
            | RETURN
            | LITINT of int
            | LITFLOAT of float
            | LITSTRING of string
            | VAR
            | LET
            | ID of string
            | NEW
            | OBJ
            | PONTO
            | DOISPTO
            | CONT
            | DO
            | CLASS
            | CONST
            | IN
            | OF
            | NULL
```

```
            | TRUE
            | FALSE
            | PROMPT
            | EOF
}

let digito = ['0' - '9']
let inteiro = digito+
let real = (digito+ '.' digito+ | '.' digito+)
let letra = ['a' - 'z' 'A' - 'Z']
let identificador = letra ( letra | digito | '_')*

let brancos = [' ' '\t']+
let novalinha = '\r' | '\n' | "\r\n"

let comentario = "//" [^ '\r' '\n' ]*

rule token = parse
  brancos       { token lexbuf }
| novalinha     { incr_num_linha lexbuf; token lexbuf }
| comentario    { token lexbuf }
| "/*"          { comentario_bloco 0 lexbuf }
| '('            { APAR }
| ')'           { FPAR }
| '{'           { ACHAVE }
| '}'           { FCHAVE }
| '['           { ACOLCH }
| ']'           { FCOLCH }
| '+'           { MAIS }
| '-'           { MENOS }
| '*'           { MULT }
| '/'           { DIV }
| '%'           { MOD }
| "**"          { POT }
| '='           { ATRIB }
| "+="          { MAISATRIB }
| "-="          { MENOSATRIB }
| "/="          { DIVATRIB }
| "*="          { MULTATRIB }
| '>'           { MAIOR }
| '<'           { MENOR }
| "<="          { MENORIGUAL }
| ">="          { MAIORIGUAL }
| "=="          { IGUAL }
| "++"          { INCR }
| "--"          { DECR }
| "&&"          { AND }
| "||"          { OR }
| '!'           { NOT }
| "!="          { DIF }
| ','           { VIRG }
| ';'           { PONTOVIRG }
| '.'           { PONTO }
| ':'           { DOISPTO }
| "for"         { FOR }
| "if"          { IF }
| "else"        { ELSE }
| "switch"      { SWITCH }
| "case"        { CASE }
```

```
| "break"       { BREAK }
| "default"     { DEFAULT }
| "console.log"   { CONSOLELOG }
| "function"        { FUNCTION }
| "return"          { RETURN }
| "var"               { VAR }
| "while"            { WHILE }
| "new"              { NEW }
| "let"                { LET }
| "Object"          { OBJ }
| "continue"       { CONT }
| "do"          { DO }
| "in"          { IN }
| "of"          { OF }
| "class"       { CLASS }
| "const"       { CONST }
| "null"        { NULL }
| "true"        { TRUE }
| "false"       { FALSE }
| "prompt"      { PROMPT }
| '"'           { let pos = lexbuf.lex_curr_p in
                  let lin = pos.pos_lnum
                  and col = pos.pos_cnum - pos.pos_bol - 1 in
                  let buffer = Buffer.create 1 in
                  let str = leia_string lin col buffer lexbuf in
                  LITSTRING str }
| "'"           {let pos = lexbuf.lex_curr_p in
                  let lin = pos.pos_lnum
                  and col = pos.pos_cnum - pos.pos_bol - 1 in
                  let buffer = Buffer.create 1 in
                  let str = leia_string_simp lin col buffer lexbuf in
                  LITSTRING str}
| inteiro as num  { let numero = int_of_string num in
                  LITINT numero  }
| real as num     { let numero = float_of_string num in
                  LITFLOAT numero  }
| identificador as id { ID id }
| _ as c  { failwith (msg_erro lexbuf c) }
| eof        { EOF }

and leia_string lin col buffer = parse
'"' { Buffer.contents buffer}
| "\\t" { Buffer.add_char buffer '\t';
leia_string lin col buffer lexbuf }
| "\\n" { Buffer.add_char buffer '\n';
leia_string lin col buffer lexbuf }
| '\\' '"' { Buffer.add_char buffer '"';
leia_string lin col buffer lexbuf }
| '\\' '\\' { Buffer.add_char buffer '\\';
leia_string lin col buffer lexbuf }
| _ as c { Buffer.add_char buffer c;
leia_string lin col buffer lexbuf }
| eof { erro lin col "A string nao foi fechada"}

and leia_string_simp lin col buffer = parse
"'" { Buffer.contents buffer}
| "\\t" { Buffer.add_char buffer '\t';
leia_string_simp lin col buffer lexbuf }
| "\\n" { Buffer.add_char buffer '\n';
```

```
leia_string_simp lin col buffer lexbuf }
| '\\' "'" { Buffer.add_char buffer '\'';
leia_string_simp lin col buffer lexbuf }
| '\\' '\\' { Buffer.add_char buffer '\\';
leia_string_simp lin col buffer lexbuf }
| _ as c { Buffer.add_char buffer c;
leia_string_simp lin col buffer lexbuf }
| eof { erro lin col "A string nao foi fechada"}

and comentario_bloco n = parse
"*/"        { if n=0 then token lexbuf
              else comentario_bloco (n-1) lexbuf }
| "/*"      { failwith "Comentarios aninhados nao permitidos" }
| novalinha { incr_num_linha lexbuf;
              comentario_bloco n lexbuf }
| _         { comentario_bloco n lexbuf }
| eof       { failwith "Comentario nao fechado" }
```

terminal

4.1 Geração do Código do Analisador Léxico

A geração do código do analisador léxico usando o Ocamllex é feita utilizando os comandos:

```
> ocamllex lexico.mll
> ocamlc -c lexico.ml
```

Após utilizar estes comandos será gerado um arquivo ”lexico.ml”, que apresenta o código do analisador léxico.

4.2 Gerando Tokens

Para testar seu funcionamento foi usado o arquivo nano12.js e para isso, utiliza-se os comandos:

```
> rlwrap ocaml
# # use "carregador.ml";;
# lex "nano12.js" ;;
```

Tokes gerados:

```
- : lexico.tokens list =
[lexico.FUNCTION; lexico.ID "main"; lexico.APAR; lexico.FPAR;
 lexico.DOISPTO; lexico.ID "void"; lexico.ATRIB; lexico.ACHAVE;
 lexico.VAR; lexico.ID "n"; lexico.DOISPTO; lexico.ID "integer";
 lexico.PONTOVIRG; lexico.VAR; lexico.ID "m"; lexico.DOISPTO;
 lexico.ID "integer"; lexico.PONTOVIRG; lexico.VAR; lexico.ID "x";
 lexico.DOISPTO; lexico.ID "integer"; lexico.PONTOVIRG;
 lexico.ID "n"; lexico.ATRIB; lexico.LITINT 1; lexico.PONTOVIRG;
 lexico.ID "m"; lexico.ATRIB; lexico.LITINT 2; lexico.PONTOVIRG;
 lexico.ID "x"; lexico.ATRIB; lexico.LITINT 5; lexico.PONTOVIRG;
 lexico.WHILE; lexico.APAR; lexico.ID "x"; lexico.MAIOR;
 lexico.ID "n"; lexico.FPAR; lexico.ACHAVE; lexico.IF; lexico.APAR;
 lexico.ID "n"; lexico.IGUAL; lexico.ATRIB; lexico.ID "m";
 lexico.FPAR; lexico.ACHAVE; lexico.CONSOLELOG; lexico.APAR;
 lexico.ID "n"; lexico.FPAR; lexico.PONTOVIRG; lexico.FCHAVE;
 lexico.ELSE; lexico.ACHAVE; lexico.CONSOLELOG; lexico.APAR;
 lexico.LITINT 0; lexico.FPAR; lexico.PONTOVIRG; lexico.FCHAVE;
 lexico.FCHAVE; lexico.FCHAVE; lexico.ID "main"; lexico.APAR;
```

```
  lexico.FPAR; lexico.PONTOVIRG; lexico.EOF]
```

5 Analisador Sintático

É o processo de analisar uma sequência de entrada para determinar sua estrutura gramatical segundo uma determinada gramática formal. Nessa etapa se determina se uma cadeia de símbolos léxicos pode ser gerada por uma gramática. O analisador sintático pode ser iplementado usadndo o algoritmo recursivo descendente(parser preditivo). Cada produção da gramática corresponde a um método recursivo.
O parser recebe os tokens gerado na análise léxica e determina se as sequências de tokens recebidas formam sentenças válidas para a linguagem para estar de acordo com a sua sintaxe. Então, o analisador sintático produz uma árvore como saída.

5.1 Códigos do Analisador sintático

Os arquivos necessários para a analise sintática são mostrados a seguir.

lexico.mll

```
{
  open Lexing
  open Printf
  open Sintatico

  exception Erro of string

(* Increment the line and column counter *)
  let incr_num_linha lexbuf =
    let pos = lexbuf.lex_curr_p in
    lexbuf.lex_curr_p <- { pos with
      pos_lnum = pos.pos_lnum + 1;
      pos_bol = pos.pos_cnum;
    }
}

let digito = ['0' - '9']
let inteiro = digito+
let real = digito* '.' digito+
let letra = ['a' - 'z' 'A' - 'Z']
let identificador = letra ( letra | digito | '_')*
let brancos = [' ' '\t']+
let novalinha = '\r' | '\n' | "\r\n" | "\\n"
let comentario = "//" [^ '\r' '\n' ]*


rule token = parse
  (*Caracteres em branco*)
  brancos { token lexbuf }
  | novalinha { incr_num_linha lexbuf; token lexbuf }
  | comentario { token lexbuf }
  | "/*"      { let pos = lexbuf.lex_curr_p in
            let lin = pos.pos_lnum
            and col = pos.pos_cnum - pos.pos_bol - 1 in
            comentario_bloco lin col 0 lexbuf
          }
```

```
| '{'      {ACHAVE}
| '}'      {FCHAVE}
| '('      { APAR }
| ')'      { FPAR }

(*Separators*)
| ','      { VIRG }
| ';'      { PONTOVIRG }
| '.'      { PONTO }

(*Reservad Words*)
| "var"      { VAR }
| "console.log"    { PRINT }
| "prompt"   { LEIA }
|"function" { FUNCAO }
|"if"        { IF }
|"else if"   { ELIFE }
|"else"      { ELSE }
|"while"     { WHILE }
|"for"       { FOR }
|"do"        { DO }
|"case"      { CASE }
|"switch"    { SWITCH }
|"break"     { BREAK }
|"continue" { CONTINUE }
|"return"    { RETURN}
|"default"     { DEFAULT}


(*Operators*)
| '+'      { MAIS }
| '-'      { MENOS }
| '/'      { DIVIDE }
| '*'      { MULTIPLICA }
| "**"     { EXPOENTE }
| '>'      { MAIOR }
| '<'      { MENOR }
|"==="       { IGUAL }
|':'       { DOISPONTOS }
|">="      { MAIORIGUAL }
|"<="      { MENORIGUAL }
|"!="      { DIFERENTE }
|"--"      { DECREMENTO }
|"++"      { INCREMENTO }
|"="       { ATRIB }
|'%'       { MODULO }
|"||"      { OU }
|"&&"      { E }

(* Tipos *)

|"void"      { VOID  }
|"integer"   { INTEIRO }
|"string"    { STRING }
|"char"      { CHAR }
|"bool"      { BOOL }

(* Literal Variables *)
|inteiro as num { let numero = int_of_string num in LITINT numero }
```

```
  |real as r    {let r = float_of_string r in LITREAL r}
  |identificador as id { ID id }
  |'"'       { let pos = lexbuf.lex_curr_p in
          let lin = pos.pos_lnum
          and col = pos.pos_cnum - pos.pos_bol - 1 in
          let buffer = Buffer.create 1 in
          let str = leia_string lin col buffer lexbuf in
          LITSTRING str }
  |_         { raise (Erro ("Caracter desconhecido: " ^ Lexing.lexeme
     lexbuf )) }
  |eof       { EOF }

(* Detect block comments *)
and comentario_bloco lin col n = parse
  "*/"       { if n=0 then token lexbuf
          else comentario_bloco lin col (n-1) lexbuf }
  | "/*"       { comentario_bloco lin col (n+1) lexbuf }
  | novalinha    { incr_num_linha lexbuf; comentario_bloco lin col n lexbuf
      }
  | _        { comentario_bloco lin col n lexbuf }
  | eof      { raise (Erro "Comentário não terminado") }

(* Identify the string and ignore special characters *)
and leia_string lin col buffer = parse
  '"'{ Buffer.contents buffer}
    | "\\t"      { Buffer.add_char buffer '\t'; leia_string lin col buffer
       lexbuf }
    | "\\n"      { Buffer.add_char buffer '\n'; leia_string lin col buffer
       lexbuf }
    | '\\' '"'   { Buffer.add_char buffer '"'; leia_string lin col buffer
       lexbuf }
    | '\\' '\\' { Buffer.add_char buffer '\\'; leia_string lin col buffer
       lexbuf }
    | _ as c     { Buffer.add_char buffer c; leia_string lin col buffer
       lexbuf}
    | eof        { raise (Erro "A string não foi fechada")}
```

Sintatico.mly

```
%{
  open Ast
%}

(* Literais *)
%token <int> LITINT
%token <bool> LITBOOL
%token <float> LITREAL
%token <string> ID
%token <string> LITSTRING
%token INTEIRO
%token STRING
%token CHAR
%token BOOL


(* Tokens de estrutura *)
%token DOISPONTOS
%token ACHAVE FCHAVE
```

```
(* Tokens de listagem e delimitação *)
%token VIRG PONTOVIRG PONTO
%token APAR FPAR

(* Token de definição de variáveis *)
%token VAR

(* Tokens de condição *)
%token IF ELSE
%token ELIFE

(* Tokens de repetição *)
%token DO
%token WHILE
%token FOR
%token DEFAULT
%token BREAK
%token CONTINUE
%token SWITCH CASE

(* Tokens de I/O *)
%token PRINT LEIA

(* Tokens de operações *)
%token ATRIB
%token MAIS MENOS
%token MULTIPLICA DIVIDE
%token INCREMENTO DECREMENTO
%token MODULO
%token VOID
%token MAIOR MENOR
%token IGUAL DIFERENTE
%token MAIORIGUAL MENORIGUAL
%token E OU
%token FUNCAO
%token RETURN
%token EOF
%token EXPOENTE

%left OU
%left E
%left IGUAL DIFERENTE
%left MAIOR MENOR
%left MAIORIGUAL MENORIGUAL
%left MAIS MENOS
%left MULTIPLICA DIVIDE MODULO

%start <Ast.programa> programa

%%

(* Os programas em javascript possuem declaração de variáveis e funções
   misturados, bem como sua execução *)
programa: fs = declaracao_funcao*
      de = declaracao*
      cs = comando*
      EOF { Programa (fs, List.flatten de,cs) }

tipo: t=tipo_simples { t }
```

```
(* Definição de tipos               - OOK *)
tipo_simples:   | LITINT    { TipoInt   }
        | INTEIRO     { TipoInt   }
        | LITREAL   { TipoReal    }
        | LITSTRING   { TipoString  }
        | BOOL      { TipoBool    }
        | VOID      { TipoVoid    }

(* Definição de parâmetros          - OOK *)

parametros: dec = separated_list(PONTOVIRG, declaracao_args) { List.
   flatten dec}

declaracao_args: ids = separated_nonempty_list(VIRG, ID) DOISPONTOS t =
   tipo {List.map (fun id -> DecVar (id,t)) ids}

(* Definição de função            - OOK *)
declaracao_funcao:
    |FUNCAO id=ID APAR p=parametros FPAR DOISPONTOS tp = tipo_simples
       ATRIB
      ACHAVE
        dec = declaracao*
        lc = comando*
      FCHAVE {Funcao(id, p, tp , List.flatten dec, lc) }


(* parametros: ids = separated_list(VIRG, option(ID)) DOISPONTOS t=
   tipo_simples {List.map (fun id -> Parametros (id,t)) ids } *)

(* DEFINIÇÃO DE COMANDOS          - OOK *)
comando: c = comando_atribuicao     { c }
    |c = comando_se         { c }
    |c = comando_entrada      { c }
    |c = comando_saida        { c }
    |c = comando_for        { c }
    |c = comando_while        { c }
    |c = comando_case       { c }
    |c = comando_funcao       { c }
    |c = comando_return       { c }
    |c = comando_return_vazio   { c }

(* Comandos return *)
comando_return: RETURN exp = expressao PONTOVIRG        {CmdRet (exp)}

comando_return_vazio: RETURN PONTOVIRG          {CmdRetv}
           |RETURN APAR FPAR PONTOVIRG    {CmdRetv}

(* Declaração de variávies          - OOK *)
declaracao: VAR ids = separated_nonempty_list(VIRG, ID) DOISPONTOS t =
   tipo PONTOVIRG {List.map (fun id -> DecVar (id,t)) ids }

(* Definição de atribuição          - OOK *)
comando_atribuicao: v = variavel ATRIB e = expressao PONTOVIRG {CmdAtrib (
   v,e)}

(* Chamdas de função            - OOK *)
comando_funcao: id = ID APAR  arg=separated_list(VIRG, expressao) FPAR
   PONTOVIRG
```

```
          {CmdChamadaFuncao (id, arg)}

(* Definição de IF              - OOK *)
comando_se: IF APAR teste = expressao FPAR ACHAVE
        entao = comando*
      FCHAVE
        senao = option(ELSE ACHAVE cs=comando* FCHAVE{cs} ) {CmdSe (teste,
            entao, senao)}
      (* Option descreve qunando é opcional ter o próximo comando *)

(* Comando entrada              - OOK *)
comando_entrada: LEIA APAR xs=expressao FPAR PONTOVIRG {CmdEntrada xs}

(* Comando saída                - OOK *)
comando_saida: PRINT APAR xs=separated_nonempty_list(VIRG, expressao) FPAR
     PONTOVIRG { CmdSaida xs }

(* Comando For                  - OOK*)
comando_for: FOR APAR v=variavel ATRIB ex=expressao PONTOVIRG e=expressao
    PONTOVIRG exp = expressao FPAR ACHAVE
              c= comando*
            FCHAVE { CmdFor(v,ex,e,exp,c) }

(* Comando WHILE                - OOK *)
comando_while: WHILE APAR teste=expressao FPAR ACHAVE c=comando* FCHAVE {
    CmdWhile(teste,c)}

(* Comando Case                 - OOK*)
comando_case: SWITCH APAR v = variavel FPAR ACHAVE
        cas = cases* FCHAVE {CmdCase(v,cas)}

cases:  CASE e = expressao DOISPONTOS c = comando* BREAK PONTOVIRG {Case(e
    ,c)}

expressao:
      | v=variavel            { ExpVar v        }
      | i=LITINT              { ExpInt i        }
      | s=LITSTRING             { ExpString s       }
      | r=LITREAL             { ExpReal r       }
      | c = comando_funcao        { ExpChamadaF c     }
      | e1=expressao op=oper e2=expressao { ExpOp (op, e1, e2)  }
      | APAR e=expressao FPAR       { Expar(e)        }

%inline oper:
      | MAIS        { Mais    }
      | MENOS       { Menos   }
      | MULTIPLICA    { Mult    }
      | DIVIDE      { Div   }
      | MODULO      { Mod   }
      | MENOR       { Menor   }
      | IGUAL       { Igual   }
      | MENORIGUAL    { MenorIgual}
      | MAIORIGUAL    { MaiorIgual}
      | DIFERENTE     { Difer   }
      | MAIOR       { Maior   }
      | E         { And     }
      | OU        { Or    }

variavel:
```

```
      | x=ID        { VarSimples x }
      | v=variavel PONTO x=ID { VarCampo (v,x) }
```

Ast.ml

```
(* The type of the abstract syntax tree (AST). *)
type ident = string

type programa = Programa of funcoes* declaracoes *  comandos
and comandos = comando list
and declaracao = DecVar of ident * tipo
and declaracoes = declaracao list

and parametros = declaracao list
and decfuncao = Funcao  of ident * parametros * tipo * declaracoes *
   comandos
and funcoes = decfuncao list

and tipo = TipoInt
          | TipoReal
          | TipoString
          | TipoChar
          | TipoBool
          | TipoVoid
and campos = campo list
and campo = ident * tipo
and cases = Case of expressao * comandos
and comando = CmdAtrib          of variavel * expressao
        | CmdSe         of expressao * comandos * comandos option
        | CmdEntrada      of expressao
        | CmdSaida        of expressoes
        | CmdEntradaln      of expressao
        | CmdSaidaln      of expressoes
        | CmdFor        of variavel * expressao * expressao* expressao *
           comandos
        | CmdWhile        of expressao * comandos
        | CmdCase         of variavel * cases list
        | CmdChamadaFuncao    of ident * expressoes
        | CmdRet          of expressao
        | CmdRetv
and variaveis = variavel list
and variavel = VarSimples of ident
              | VarCampo of variavel * ident

and expressao =   ExpVar of variavel
        | ExpInt of int
        | ExpString of string
        | ExpBool of bool
        | ExpReal of float
        | ExpOp of oper * expressao * expressao
        | Expar of expressao
        | ExpChamadaF of comando

and expressoes = expressao list
and oper =    Mais
      | Menos
      | Mult
      | Div
      | Mod
      | Menor
```

```
      | Igual
      | Difer
      | Maior
      | MaiorIgual
      | MenorIgual
      | And
      | Or
```

sintaticoTest.ml

```
open Printf
open Lexing
open Ast
open ErroSint (* nome do módulo contendo as mensagens de erro *)

exception Erro_Sintatico of string

module S = MenhirLib.General (* Streams *)
module I = Sintatico.MenhirInterpreter

let posicao lexbuf =
    let pos = lexbuf.lex_curr_p in
    let lin = pos.pos_lnum
    and col = pos.pos_cnum - pos.pos_bol - 1 in
    sprintf "linha %d, coluna %d" lin col

(* [pilha checkpoint] extrai a pilha do autômato LR(1) contida em
   checkpoint *)
let pilha checkpoint =   match checkpoint with
            | I.HandlingError amb -> I.stack amb
            | _ -> assert false (* Isso não pode acontecer *)


let estado checkpoint : int = match Lazy.force (pilha checkpoint) with
            | S.Nil -> 0(* O parser está no estado inicial *)
            | S.Cons (I.Element (s, _, _, _), _) -> I.number s
let sucesso v = Some v

let falha lexbuf (checkpoint : Ast.programa I.checkpoint) =
            let estado_atual = estado checkpoint in
            let msg = message estado_atual in
            raise (Erro_Sintatico (Printf.sprintf "%d - %s.\n" (Lexing.
               lexeme_start lexbuf) msg))

let loop lexbuf resultado =
            let fornecedor = I.lexer_lexbuf_to_supplier Lexico.token
               lexbuf in
            I.loop_handle sucesso (falha lexbuf) fornecedor resultado

let parse_com_erro lexbuf =
        try
          Some (loop lexbuf (Sintatico.Incremental.programa lexbuf.
             lex_curr_p))
        with
          | Lexico.Erro msg -> printf "Erro lexico na %s:\n\t%s\n" (
             posicao lexbuf) msg; None
          | Erro_Sintatico msg -> printf "Erro sintático na %s %s\n" (
             posicao lexbuf) msg; None

let parse s =
```

```
        let lexbuf = Lexing.from_string s in
        let ast = parse_com_erro lexbuf in ast

let parse_arq nome =  let ic = open_in nome in
            let lexbuf = Lexing.from_channel ic in
            let result = parse_com_erro lexbuf in
            let _ = close_in ic in
            match result with
                  | Some ast -> ast
                  | None -> failwith "A analise sintatica falhou"
```

5.1.1 Observações

é importante compilar os arquivos para que os comandos usados no proximo passo executem corretamente.
Exemplo: Ast.ml
Para compila-lo basta chamar no terminal:

```
ocamlc -c ast.ml
```

O mesmo deve ser feito para os outros arquivos.

5.2 Menhir

Para implementar um analisador sintático para JavaScript é utilziado o Menhir, um gerador de analisadores sintáticos LR(1).
Primeiro são definidas as sentenças válidas da linguagem JavaScript. Também é necessário definir mensagens de erro, para os erros de sintaxe, que pode ser gerado pelo Menhir.
Observação: Caso o Menhir não esteja instalado, o mesmo pode ser feito através do opam:

```
opam install menhir
```

Houve necessidade de reinstalar o ocaml, para isso é necessário remover o opam e suas bibliotecas primeiro.

```
sudo apt-get purge --auto-remove opam
```

Depois basta repetir a instalação, como mostra a seção 0.2.2.

5.2.1 Passos para gerar e compilar

Devemos informar o Ocamlbuild que é para usar o Menhir, para que o mesmo não use o OcamlYaac.

```
ocamlbuild -use-menhir main.byte
```

Gerando o analisador sintatico

```
menhir sintatico.mly
```

Dois arquivos são gerados, sintatico.mli (interface com as definições dos tokens) e o sintatico.ml (analisador sintatico LR(1).

Gerando o analisador léxico

```
ocamllex lexico.mll
```

Que gera o arquivo lexico.ml

Compilando a interface, caso não tenha compilado o arquivo "ast.ml", o momento é esse, pois o mesmo é necessário para a compilação da interface.

```
ocamlc -c sintatico.mli
```

Que gera o sintatico.cmi que é a interface compilada.

Devemos compilar também o arquivo do analisador sintatico e do analisador lexico.

```
ocamlc -c sintatico.ml

ocamlc -c lexico.ml
```

Que geram os arquivos sintatico.cmo e lexico.cmo, respectivamente.

5.2.2 Arquivo de inicialização

O arquivo .ocamlinti inicialisa os arquivos necessários para a compilação de forma rapida. .ocamlinit

```
let () =
    try Topdirs.dir_directory (Sys.getenv "OCAML_TOPLEVEL_PATH")
    with Not_found -> ()
;;

#use "topfind";;
#require "menhirLib";;
#directory "_build";;

#load "Ast.cmo";;
#load "erroSint.cmo";;
#load "Lexico.cmo";;
#load "Sintatico.cmo";;

#load "sintaticoTest.cmo";;
open Ast
open SintaticoTest
```

5.2.3 Tratando erros

Primeiro, todos os arquivos compilados anteriormente deve ser excluidos, dessa forma não interferem na compilação do projeto todo.

```
rm -f *.cmi *cmo lexico.ml sintatico.ml sintatico.mli
```

Para gerar o arquivo sintatico com os erros basta o seguinte comando e editar as mensagens de erro que se encontram no arquivo:

```
menhir -v --list-errors sintatico.mly > sintatico.msg
```

Após editar as mensagens de erro basta compilar o arquivo com os erros e compilar o analisador sintático através dos comandos:

```
menhir sintatico.mly --compile-errors sintatico.msg > erroSint.ml
```

5.3 Compilação:

Para compilar todo o projeto com o ocamlbuild:

```
ocamlbuild -use-ocamlfind -use-menhir -menhir " menhir --table " -package
    menhirLib sintaticoTest.byte
```

para facilitar a compilação, um .sh se faz necessário.
run.sh

```
#!/bin/bash
rm -f *.cmi *cmo lexico.ml sintatico.ml sintatico.mli

menhir -v --list-errors Sintatico.mly > Sintatico.msg;

menhir Sintatico.mly --compile-errors Sintatico.msg > erroSint.ml;

ocamlbuild -use-ocamlfind -use-menhir -menhir "menhir --table" -package
    menhirLib sintaticoTest.byte;

rlwrap ocaml;

# para dar permissao pro script rodar digita no terminal chmod 777 script.
    sh
# para executar: bash script.sh
```

Ao executar o script acima todo projeto é compilado e o ocmal é iniciado, para testar o analisador sintatico basta:

```
parse_arq "nano12.js";;
```

5.3.1 Árvore Sintática Abstrata

Com isso a árvore sintática abstrata desse programa .js testado é gerada.

```
- : Ast.programa option =
Some
 (Programa
   ([Funcao ("main", [], TipoVoid,
      [DecVar ("n", TipoInt); DecVar ("m", TipoInt); DecVar ("x", TipoInt)
         ],
      [CmdAtrib (VarSimples "n", ExpInt 1);
       CmdAtrib (VarSimples "m", ExpInt 2);
       CmdAtrib (VarSimples "x", ExpInt 5);
       CmdWhile
        (ExpOp (Maior, ExpVar (VarSimples "x"), ExpVar (VarSimples "n")),
        [CmdSe
          (ExpOp (Igual, ExpVar (VarSimples "n"), ExpVar (VarSimples "m"))
             ,
          [CmdSaida [ExpVar (VarSimples "n")];
           CmdChamadaFuncao ("print",
            [ExpString "%d"; ExpVar (VarSimples "n")])],
          Some
           [CmdSaida [ExpInt 0];
            CmdChamadaFuncao ("printf",
             [ExpString "%d"; ExpVar (VarSimples "n")])])])])],
   [], [CmdChamadaFuncao ("main", [])]))
```

6 Analisador Semântico

A partir da árvore sintática e a tabela de símbolos é feita a analise semântica. A analise semântica é responsavel pela verificação de aspectos relacionados aos significados das instruções, nessa etapa é feita a validação de uma séire de regras que não podem ser verificadas nas outras etapas da compilação.
A analise semântica percorre a árvore sintática e relaciona os identificadores com seus dependentes de acordo com a estrutura hierarquica, também captura informações sobre o programa fonte para que as fases subsequentes possam gerar o código objeto, um importante compontente é a verificação de tipos, que é feita nesta etapa, nela o compilador verifica se cada operador recebe os operandos permitidos e especificados na linguagem fonte.
Assim como a analise sintatica, ao fim desta eta será gerada uma árvore, mas uma árvore tipada. A implementação desta etapa tamém é feita através do Menhir. A seguir são apresentadas as etapas para a implementação, compilação e teste do analisador semântico.

6.1 Especificação semântica para JavaScript

Nessa seção será apresentados os arquivos necessarios para a implementação do analisador semântico. É importante ressaltar que foram realizadas algumas alterações nos arquivos correspondentes às etapas anteriores.

sintatico.mly

```
%{
open Lexing
open Ast
open Sast
%}

/* Literais */
%token <int * Lexing.position> LITINT
%token <bool * Lexing.position> LITBOOL
%token <float * Lexing.position> LITREAL
%token <string * Lexing.position> ID
%token <string * Lexing.position> LITSTRING
%token <char * Lexing.position> LITCHAR
%token <Lexing.position> INTEIRO
%token <Lexing.position> STRING
%token <Lexing.position> CHAR
%token <Lexing.position> BOOL
%token <Lexing.position> REAL


/* Tokens de estrutura */
%token <Lexing.position> DOISPONTOS
%token <Lexing.position> ACHAVE FCHAVE


/* Tokens de listagem e delimitação */
%token <Lexing.position> VIRG PONTOVIRG
%token <Lexing.position> APAR FPAR

/* Token de definição de variáveis */
%token <Lexing.position> VAR

/* Tokens de condição */
```

```
%token <Lexing.position> IF ELSE

/* Tokens de repetição */
%token <Lexing.position> WHILE
%token <Lexing.position> FOR
%token <Lexing.position> DEFAULT
%token <Lexing.position> BREAK
%token <Lexing.position> SWITCH CASE

/* Tokens de I/O */
%token <Lexing.position> PRINT LEIA

/* Tokens de operações */
%token <Lexing.position> ATRIB
%token <Lexing.position> MAIS MENOS
%token <Lexing.position> MULTIPLICA DIVIDE
%token <Lexing.position> INCREMENTO DECREMENTO
%token <Lexing.position> MODULO
%token <Lexing.position> VOID
%token <Lexing.position> MAIOR MENOR
%token <Lexing.position> IGUAL DIFERENTE
%token <Lexing.position> MAIORIGUAL MENORIGUAL
%token <Lexing.position> E OU
%token <Lexing.position> FUNCAO
%token <Lexing.position> RETURN
%token EOF
%token <Lexing.position> EXPOENTE

%left OU
%left E
%left IGUAL DIFERENTE
%left MAIOR MENOR
%left MAIORIGUAL MENORIGUAL
%left MAIS MENOS
%left MULTIPLICA DIVIDE MODULO EXPOENTE


%start <Sast.expressao Ast.programa> programa

%%

/* Os programas em javascript possuem declaração de variáveis e funções
   misturados, bem como sua execução */
programa: de = declaracoes
      fs = declaracao_funcao+
      c = comando*
      EOF { Programa (de, fs, c) }


/* Definição de tipos               - OOK */
tipo_simples:
        | INTEIRO     { TipoInt     }
        | REAL        { TipoReal    }
            | CHAR        { TipoChar      }
        | STRING      { TipoString  }
        | BOOL      { TipoBool    }
        | VOID      { TipoVoid    }

/* Definição de parâmetros          - OOK */
```

```
parametros: dec = separated_list(VIRG, declaracao_args) { List.flatten dec
    }

declaracao_args: ids = separated_nonempty_list(VIRG, ID)
 DOISPONTOS t = tipo_simples
 {   List.map (fun id -> (id,t)) ids}

/* Definição de função            - OOK */
declaracao_funcao:
    FUNCAO id=ID APAR p=parametros FPAR DOISPONTOS tp = tipo_simples ATRIB
      ACHAVE
        dec = declaracoes
        lc = comando*
      FCHAVE {
          Funcao{
  fn_nome=      id;
  fn_tiporet=   tp;
  fn_formais=   p;
  fn_locais=   dec;
  fn_corpo=   lc
          }
}



/* parametros: ids = separated_list(VIRG, option(ID)) DOISPONTOS t=
   tipo_simples {List.map (fun id -> Parametros (id,t)) ids } */

/* DEFINIÇÃO DE COMANDOS          - OOK */
comando: c = comando_atribuicao     { c }
    |c = comando_se             { c }
    |c = comando_entrada          { c }
    |c = comando_saida          { c }
    |c = comando_for            { c }
    |c = comando_while        { c }
    |c = comando_case         { c }
    |c = comando_expressao        { c }
    |c = comando_return         { c }
    |c = comando_return_vazio   { c }
    |c = comando_incremento     { c }
    |c = comando_decremento     { c }

/* Comandos return */
comando_return: RETURN exp = expressao PONTOVIRG    { CmdRetorno (Some exp
    )}

comando_return_vazio: RETURN PONTOVIRG          { CmdRetorno None }
           |RETURN APAR FPAR PONTOVIRG    { CmdRetorno None   }

/* Declaração de variávies          - OOK */
declaracoes: dec = declaracao* { List.flatten dec}
declaracao: VAR ids = separated_nonempty_list(VIRG, ID) DOISPONTOS t =
   tipo_simples PONTOVIRG {List.map (fun id -> DecVar (id,t)) ids }

/* Definição de atribuição          - OOK */
comando_atribuicao: v = variavel ATRIB e = expressao PONTOVIRG {CmdAtrib (
   ExpVar v,e)}
```

```
/* Comando Expressão            - OOK */
comando_expressao: e = expressao PONTOVIRG {  ComandoExpress (e) }


/* Definição de IF              - OOK */
comando_se: IF APAR teste = expressao FPAR ACHAVE
        entao = comando*
      FCHAVE
        senao = option(ELSE ACHAVE cs=comando* FCHAVE{cs} ) {CmdSe (teste,
            entao, senao)}
      /* Option descreve qunando é opcional ter o próximo comando */

/* Comando entrada              - OOK */
comando_entrada: LEIA APAR xs=expressao FPAR PONTOVIRG {CmdEntrada xs}

/* Comando saída                - OOK */
comando_saida: PRINT APAR xs=separated_nonempty_list(VIRG, expressao) FPAR
    PONTOVIRG { CmdSaida xs }

/* Comando For                  - OOK*/
comando_for: FOR APAR v=variavel ATRIB ex=expressao PONTOVIRG e=expressao
   PONTOVIRG exp = expressao FPAR ACHAVE
              c= comando*
            FCHAVE { CmdFor(ExpVar v,ex,e,exp,c) }

/* Comando WHILE                - OOK */
comando_while: WHILE APAR teste=expressao FPAR ACHAVE c=comando* FCHAVE {
   CmdWhile(teste,c)}

/* Comando Case                 - OOK*/
comando_case: SWITCH APAR v = variavel FPAR ACHAVE
        cas = cases*
        def = option(case_default)
        FCHAVE {CmdCase(ExpVar v,cas, def)}

cases:  CASE e = expressao DOISPONTOS c = comando* BREAK PONTOVIRG  { Case
   (e,c) }
case_default: DEFAULT DOISPONTOS c = comando* BREAK PONTOVIRG     { c }

/* Comando DECREMENTO             - OOK */
comando_decremento: x=variavel dec=DECREMENTO PONTOVIRG {
  CmdAtrib ((ExpVar x),
  ExpOp ((Menos,dec), ExpVar(x), ExpInt(1,  dec)))
  }

/* Comando INCREMENTO             - OOK */
comando_incremento: x=variavel inc =INCREMENTO PONTOVIRG {
  CmdAtrib ((ExpVar x),
  ExpOp ((Mais,inc), ExpVar(x), ExpInt(1,  inc)))
  }

expressao:
      | v=variavel            { ExpVar v        }
      | i=LITINT              { ExpInt i        }
      | s=LITSTRING             { ExpString s       }
      | r=LITREAL             { ExpReal r       }
      | c=LITCHAR             { ExpChar c       }
      | b=LITBOOL             { ExpBool b       }
      | e1=expressao op=oper e2=expressao { ExpOp (op, e1, e2)  }
```

```
      | APAR e=expressao FPAR       { e           }
      | c=chamada         { c           }

chamada: x=ID APAR args=separated_list(VIRG,expressao) FPAR { ExpChamada (
    x,args) }

%inline oper:
      | pos = MAIS    { (Mais, pos)     }
      | pos = MENOS     { (Menos, pos)    }
      | pos = MULTIPLICA  { (Mult, pos)     }
      | pos = DIVIDE    { (Div, pos)    }
      | pos = MODULO    { (Mod, pos)    }
      | pos = MENOR     { (Menor, pos)    }
      | pos = IGUAL     { (Igual, pos)    }
      | pos = MENORIGUAL  { (MenorIgual, pos) }
      | pos = MAIORIGUAL  { (MaiorIgual, pos) }
      | pos = DIFERENTE   { (Difer, pos)    }
      | pos = MAIOR     { (Maior, pos)    }
      | pos = E       { (And, pos)    }
      | pos = OU      { (Or, pos)     }
      | pos = EXPOENTE  { (Expoente, pos) }
variavel:
      | x=ID        { VarSimples x }
```

lexico.mll

```
{
  open Lexing
  open Printf
  open Sintatico

  exception Erro of string

  let incr_num_linha lexbuf =
    let pos = lexbuf.lex_curr_p in
    lexbuf.lex_curr_p <-
      { pos with pos_lnum = pos.pos_lnum + 1;
                 pos_bol = pos.pos_cnum
      }

  let pos_atual lexbuf = lexbuf.lex_start_p

}

let digito = ['0' - '9']
let inteiro = digito+
let real = digito* '.' digito+
let letra = ['a' - 'z' 'A' - 'Z']
let identificador = letra ( letra | digito | '_')*

let brancos = [' ' '\t']+
let novalinha = '\r' | '\n' | "\r\n"

let comentario = "//" [^ '\r' '\n' ]*

rule token =
  parse
  (*Caracteres em branco*)
  brancos { token lexbuf }
  | novalinha { incr_num_linha lexbuf; token lexbuf }
```

```
| comentario { token lexbuf }
| "/*"      { let pos = lexbuf.lex_curr_p in
          let lin = pos.pos_lnum
          and col = pos.pos_cnum - pos.pos_bol - 1 in
          comentario_bloco lin col 0 lexbuf
        }
| '{'     {ACHAVE (pos_atual lexbuf) }
| '}'     {FCHAVE (pos_atual lexbuf) }
| '('     { APAR (pos_atual lexbuf) }
| ')'     { FPAR (pos_atual lexbuf) }

(*Separators*)
| ','     { VIRG (pos_atual lexbuf) }
| ';'     { PONTOVIRG (pos_atual lexbuf) }

(*Reservad Words*)
| "var"     { VAR (pos_atual lexbuf) }
| "console.log"   { PRINT (pos_atual lexbuf) }
| "readline"  { LEIA (pos_atual lexbuf) }
|"function" { FUNCAO (pos_atual lexbuf) }
|"if"       { IF (pos_atual lexbuf) }
|"else"     { ELSE (pos_atual lexbuf) }
|"while"    { WHILE (pos_atual lexbuf) }
|"for"      { FOR (pos_atual lexbuf) }
|"case"     { CASE (pos_atual lexbuf) }
|"switch"   { SWITCH (pos_atual lexbuf) }
|"break"    { BREAK (pos_atual lexbuf) }
|"return"   { RETURN  (pos_atual lexbuf) }
|"default"    { DEFAULT (pos_atual lexbuf) }


(*Operators*)
| '+'     { MAIS  (pos_atual lexbuf) }
| '-'     { MENOS  (pos_atual lexbuf) }
| '/'     { DIVIDE  (pos_atual lexbuf) }
| '*'     { MULTIPLICA (pos_atual lexbuf) }
| "**"    { EXPOENTE (pos_atual lexbuf) }
| '>'     { MAIOR (pos_atual lexbuf) }
| '<'     { MENOR (pos_atual lexbuf) }
|"==="      { IGUAL (pos_atual lexbuf) }
|':'      { DOISPONTOS (pos_atual lexbuf) }
|">="     { MAIORIGUAL (pos_atual lexbuf) }
|"<="     { MENORIGUAL (pos_atual lexbuf) }
|"!="     { DIFERENTE (pos_atual lexbuf) }
|"--"     { DECREMENTO (pos_atual lexbuf) }
|"++"     { INCREMENTO (pos_atual lexbuf) }
|"="      { ATRIB (pos_atual lexbuf) }
|'%'      { MODULO (pos_atual lexbuf) }
|"||"     { OU (pos_atual lexbuf) }
|"&&"     { E (pos_atual lexbuf) }

(* Tipos *)

|"void"     { VOID (pos_atual lexbuf)     }
|"integer"  { INTEIRO (pos_atual lexbuf)  }
|"float"    { REAL (pos_atual lexbuf)     }
|"string"   { STRING (pos_atual lexbuf)   }
|"char"     { CHAR (pos_atual lexbuf)     }
|"bool"     { BOOL (pos_atual lexbuf)     }
```

```
  (* Literal Variables *)
  |inteiro as num { let numero = int_of_string num in LITINT (numero,
     pos_atual lexbuf) }
  |real as r    {let r = float_of_string r in LITREAL (r, pos_atual lexbuf
     ) }
  |identificador as id { ID (id, pos_atual lexbuf) }
  |'"'      { let pos = lexbuf.lex_curr_p in
          let lin = pos.pos_lnum
          and col = pos.pos_cnum - pos.pos_bol - 1 in
          let buffer = Buffer.create 1 in
          let str = leia_string lin col buffer lexbuf in if ( String.
             length(str) > 1 ) then
          LITSTRING (str, pos_atual lexbuf)
          else LITCHAR (str.[0], pos_atual lexbuf) }
  |_        { raise (Erro ("Caracter desconhecido: " ^ Lexing.lexeme
     lexbuf )) }
  |eof      { EOF }

(* Detect block comments *)
and comentario_bloco lin col n = parse
  "*/"      { if n=0 then token lexbuf
          else comentario_bloco lin col (n-1) lexbuf }
  | "/*"      { comentario_bloco lin col (n+1) lexbuf }
  | novalinha   { incr_num_linha lexbuf; comentario_bloco lin col n lexbuf
      }
  | _       { comentario_bloco lin col n lexbuf }
  | eof     { raise (Erro "Comentário não terminado") }

(* Identify the string and ignore special characters *)
and leia_string lin col buffer = parse
  '"'{ Buffer.contents buffer}
    | "\\t"     { Buffer.add_char buffer '\t'; leia_string lin col buffer
       lexbuf }
    | "\\n"     { Buffer.add_char buffer '\n'; leia_string lin col buffer
       lexbuf }
    | '\\' '"'  { Buffer.add_char buffer '"'; leia_string lin col buffer
       lexbuf }
    | '\\' '\\' { Buffer.add_char buffer '\\'; leia_string lin col buffer
       lexbuf }
    | _ as c    { Buffer.add_char buffer c; leia_string lin col buffer
       lexbuf}
    | eof       { raise (Erro "A string não foi fechada")}
```

ast.ml

```
(* The type of the abstract syntax tree (AST). *)
open Lexing

type ident = string
type 'a pos =  'a * Lexing.position (* tipo e posição no arquivo fonte *)

type 'expr programa = Programa of declaracoes * ('expr funcoes) * ('expr
   comandos)
and declaracoes = declaracao list
and 'expr funcoes = ('expr funcao) list
and 'expr comandos = ('expr comando) list

and declaracao = DecVar of (ident pos) * tipo
```

```
and 'expr funcao = Funcao of ('expr decfn)

and 'expr decfn =  {
  fn_nome:    ident pos;
  fn_tiporet: tipo;
  fn_formais: (ident pos * tipo) list;
  fn_locais:  declaracoes;
  fn_corpo:   'expr comandos
}

and tipo = TipoInt
         | TipoString
         | TipoBool
         | TipoReal
         | TipoChar
         | TipoVoid
         | TipoArranjo of tipo * (int pos) * (int pos)
         | TipoRegistro of campos

and campos = campo list
and campo = ident pos * tipo

and 'expr comando =
  | CmdAtrib of 'expr * 'expr
  | CmdSe of 'expr * ('expr comandos) * ('expr comandos option)
  | CmdEntrada of ('expr )
  | CmdSaida of ('expr expressoes)
  | CmdRetorno of 'expr option
  | CmdWhile of 'expr * ('expr comandos)
  | CmdFor of   'expr * 'expr * 'expr * 'expr * ('expr comandos)
  | ComandoExpress of   'expr
  | CmdCase of  'expr * ('expr case) list * ('expr comandos option)


and 'expr variaveis = ('expr variavel) list
and 'expr variavel =
  | VarSimples of ident pos

and 'expr expressoes = 'expr list

and oper =
  | Mais
  | Menos
  | Mult
  | Div
  | Menor
  | MenorIgual
  | Igual
  | MaiorIgual
  | Difer
  | Maior
  | And
  | Or
  | Concat
  | Mod
  | Expoente

and 'expr case  = Case of 'expr * 'expr comandos
```

O arquivo sast separa expressão do código da árvore sintatica abstrata.
sast.ml

```
open Ast

type expressao =
  | ExpVar of (expressao variavel)
  | ExpInt of int pos
  | ExpReal of float pos
  | ExpString of string pos
  | ExpChar of char pos
  | ExpBool of bool pos
  | ExpOp of oper pos * expressao * expressao
  | ExpChamada of ident pos * (expressao expressoes)
```

.ocamlinit

```
let () =
  try Topdirs.dir_directory (Sys.getenv "OCAML_TOPLEVEL_PATH")
  with Not_found -> ()
;;

#use "topfind";;
#require "menhirLib";;
#directory "_build";;
#load "sintatico.cmo";;
#load "lexico.cmo";;
#load "ast.cmo";;
#load "sast.cmo";;
#load "tast.cmo";;
#load "tabsimb.cmo";;
#load "ambiente.cmo";;
#load "semantico.cmo";;
#load "semanticoTest.cmo";;

open Ast
open Ambiente
open SemanticoTest
```

6.1.1 Arquivos especificos do analisador semântico

semantico.ml

```
module Amb = Ambiente
module A = Ast
module S = Sast
module T = Tast

let rec posicao exp = let open S in
  match exp with
  | ExpVar v -> (match v with
      | A.VarSimples (_,pos) -> pos
    )
  | ExpInt (_,pos) -> pos
  | ExpString  (_,pos) -> pos
  | ExpBool (_,pos) -> pos
  | ExpChar (_,pos) -> pos
  | ExpReal (_,pos) -> pos
  | ExpOp ((_,pos),_,_)  -> pos
```

```
  | ExpChamada ((_,pos), _) -> pos

type classe_op = Aritmetico | Relacional | Logico | Cadeia

let classifica op =
  let open A in
  match op with
    Or
  | And  -> Logico
  | Menor
  | Maior
  | Igual
  | MenorIgual
  | MaiorIgual
  | Difer -> Relacional
  | Mais
  | Menos
  | Mult
  | Mod
  | Expoente
  | Div -> Aritmetico
  | Concat -> Cadeia

let msg_erro_pos pos msg =
  let open Lexing in
  let lin = pos.pos_lnum
  and col = pos.pos_cnum - pos.pos_bol - 1 in
  Printf.sprintf "Semantico -> linha %d, coluna %d: %s" lin col msg

let msg_erro nome msg =
  let pos = snd nome in
  msg_erro_pos pos msg

let nome_tipo t =
  let open A in
    match t with
      TipoInt -> "inteiro"
    | TipoString -> "string"
    | TipoBool -> "bool"
    | TipoReal -> "float"
    | TipoChar -> "char"
    | TipoVoid -> "void"
    | TipoArranjo (t,i,f) -> "arranjo"
    | TipoRegistro cs -> "registro"

let mesmo_tipo pos msg tinf tdec =
  if tinf <> tdec
  then
    let msg = Printf.sprintf msg (nome_tipo tinf) (nome_tipo tdec) in
    failwith (msg_erro_pos pos msg)

let rec infere_exp amb exp =
  match exp with
    S.ExpInt n    -> (T.ExpInt (fst n, A.TipoInt),       A.TipoInt)
  | S.ExpString s -> (T.ExpString (fst s, A.TipoString), A.TipoString)
  | S.ExpBool b   -> (T.ExpBool (fst b, A.TipoBool),     A.TipoBool)
  | S.ExpReal r   -> (T.ExpReal (fst r, A.TipoReal),     A.TipoReal)
  | S.ExpChar c   -> (T.ExpChar (fst c, A.TipoChar),     A.TipoChar)
  | S.ExpVar v ->
```

```
    (match v with
       A.VarSimples nome ->
       (* Tenta encontrar a definição da variável no escopo local, se não
             *)
       (* encontar tenta novamente no escopo que engloba o atual.
          Prossegue-se *)
       (* assim até encontrar a definição em algum escopo englobante ou at
          é    *)
       (* encontrar o escopo global. Se em algum lugar for encontrado,
                *)
       (* devolve-se a definição. Em caso contrário, devolve uma exceção
               *)
       let id = fst nome in
         (try (match (Amb.busca amb id) with
               | Amb.EntVar tipo -> (T.ExpVar (A.VarSimples nome, tipo),
                  tipo)
               | Amb.EntFun _ ->
                 let msg = "nome de funcao usado como nome de variavel: "
                    ^ id in
                  failwith (msg_erro nome msg)
           )
          with Not_found ->
                 let msg = "A variavel " ^ id ^ " nao foi declarada" in
                 failwith (msg_erro nome msg)
         )
     (* | _ -> failwith "infere_exp: não implementado" *)
    )
  | S.ExpOp (op, esq, dir) ->
    let (esq, tesq) = infere_exp amb esq
    and (dir, tdir) = infere_exp amb dir in

    let verifica_aritmetico () =
      (match tesq with
         A.TipoInt ->
         let _ = mesmo_tipo (snd op)
                      "O operando esquerdo eh do tipo %s mas o direito eh
                         do tipo %s"
                      tesq tdir
         in tesq (* O tipo da expressão aritmética como um todo *)

       | t -> let msg = "um operador aritmetico nao pode ser usado com o
          tipo " ^
                          (nome_tipo t)
         in failwith (msg_erro_pos (snd op) msg)
      )

    and verifica_relacional () =
      (match tesq with
         A.TipoInt
       | A.TipoString ->
         let _ = mesmo_tipo (snd op)
                    "O operando esquerdo eh do tipo %s mas o direito eh do
                       tipo %s"
                    tesq tdir
         in A.TipoBool (* O tipo da expressão relacional é sempre booleano
              *)

       | t -> let msg = "um operador relacional nao pode ser usado com o
          tipo " ^
```

```
                        (nome_tipo t)
          in failwith (msg_erro_pos (snd op) msg)
      )

    and verifica_logico () =
      (match tesq with
         A.TipoBool ->
         let _ = mesmo_tipo (snd op)
                   "O operando esquerdo eh do tipo %s mas o direito eh do
                      tipo %s"
                   tesq tdir
         in A.TipoBool (* O tipo da expressão lógica é sempre booleano *)

       | t -> let msg = "um operador logico nao pode ser usado com o tipo
          " ^
                        (nome_tipo t)
              in failwith (msg_erro_pos (snd op) msg)
      )
    and verifica_cadeia () =
      (match tesq with
         A.TipoString ->
         let _ = mesmo_tipo (snd op)
                   "O operando esquerdo eh do tipo %s mas o direito eh do
                      tipo %s"
                   tesq tdir
         in A.TipoString (* O tipo da expressão relacional é sempre string
             *)

       | t -> let msg = "um operador relacional nao pode ser usado com o
          tipo " ^
                        (nome_tipo t)
              in failwith (msg_erro_pos (snd op) msg)
      )

    in
    let op = fst op in
    let tinf = (match (classifica op) with
          Aritmetico -> verifica_aritmetico ()
        | Relacional -> verifica_relacional ()
        | Logico -> verifica_logico ()
        | Cadeia -> verifica_cadeia ()
      )
    in
      (T.ExpOp ((op,tinf), (esq, tesq), (dir, tdir)), tinf)

  | S.ExpChamada (nome, args) ->
     let rec verifica_parametros ags ps fs =
        match (ags, ps, fs) with
          (a::ags), (p::ps), (f::fs) ->
            let _ = mesmo_tipo (posicao a)
                      "O parametro eh do tipo %s mas deveria ser do tipo %s
                         " p f
            in verifica_parametros ags ps fs
        | [], [], [] -> ()
        | _ -> failwith (msg_erro nome "Numero incorreto de parametros")
     in
     let id = fst nome in
     try
       begin
```

```
        let open Amb in

        match (Amb.busca amb id) with
        (* verifica se 'nome' está associada a uma função *)
          Amb.EntFun {tipo_fn; formais} ->
          (* Infere o tipo de cada um dos argumentos *)
          let argst = List.map (infere_exp amb) args
          (* Obtem o tipo de cada parâmetro formal *)
          and tipos_formais = List.map snd formais in
          (* Verifica se o tipo de cada argumento confere com o tipo
             declarado *)
          (* do parâmetro formal correspondente.
                                              *)
          let _ = verifica_parametros args (List.map snd argst)
             tipos_formais
           in (T.ExpChamada (id, (List.map fst argst), tipo_fn), tipo_fn)
        | Amb.EntVar _ -> (* Se estiver associada a uma variável, falhe
            *)
          let msg = id ^ " eh uma variavel e nao uma funcao" in
          failwith (msg_erro nome msg)
      end
    with Not_found ->
      let msg = "Nao existe a funcao de nome " ^ id in
      failwith (msg_erro nome msg)

let rec verifica_cmd amb tiporet cmd =
  let open A in
  match cmd with
    CmdRetorno exp ->
    (match exp with
     (* Se a função não retornar nada, verifica se ela foi declarada como
        void *)
       None ->
       let _ = mesmo_tipo (Lexing.dummy_pos)
                  "O tipo retornado eh %s mas foi declarado como %s"
                  TipoVoid tiporet
       in CmdRetorno None
     | Some e ->
       (* Verifica se o tipo inferido para a expressão de retorno confere
          com o *)
       (* tipo declarado para a função.
                                                    *)
          let (e1,tinf) = infere_exp amb e in
          let _ = mesmo_tipo (posicao e)
                             "O tipo retornado eh %s mas foi declarado
                                como %s"
                             tinf tiporet
          in CmdRetorno (Some e1)
      )
  | CmdSe (teste, entao, senao) ->
    let (teste1,tinf) = infere_exp amb teste in
    (* O tipo inferido para a expressão 'teste' do condicional deve ser
       booleano *)
    let _ = mesmo_tipo (posicao teste)
              "O teste do if deveria ser do tipo %s e nao %s"
              TipoBool tinf in
    (* Verifica a validade de cada comando do bloco 'então' *)
    let entao1 = List.map (verifica_cmd amb tiporet) entao in
    (* Verifica a validade de cada comando do bloco 'senão', se houver *)
```

```
    let senao1 =
        match senao with
          None -> None
        | Some bloco -> Some (List.map (verifica_cmd amb tiporet) bloco)
     in
     CmdSe (teste1, entao1, senao1)

  | CmdAtrib (elem, exp) ->
    (* Infere o tipo da expressão no lado direito da atribuição *)
    let (exp,  tdir) = infere_exp amb exp
    (* Faz o mesmo para o lado esquerdo *)
    and (elem1, tesq) = infere_exp amb elem in
    (* Os dois tipos devem ser iguais *)
    let _ = mesmo_tipo (posicao elem)
                       "Atribuicao com tipos diferentes: %s = %s" tesq
                          tdir
    in CmdAtrib (elem1, exp)

  | CmdEntrada exp ->
    (* Verifica o tipo do argumento da função 'entrada' *)
    let exp = infere_exp amb exp in
    CmdEntrada ( fst exp)

  | CmdSaida exps ->
    (* Verifica o tipo de cada argumento da função 'saida' *)
    let exps = List.map (infere_exp amb) exps in
    CmdSaida (List.map fst exps)

  | CmdWhile (teste,c) ->
    let (teste1,tinf) = infere_exp amb teste in
    (* O tipo inferido para a expressão 'teste' do condicional deve ser
       booleano *)
    let _ = mesmo_tipo (posicao teste)
             "O teste do while deveria ser do tipo %s e nao %s"
             TipoBool tinf in
    (* Verifica a validade de cada comando do bloco  *)
    let c1 = List.map (verifica_cmd amb tiporet) c in

     CmdWhile (teste1, c1)

  | CmdFor ( v,ex,e,exp,c)->

    (* Infere o tipo da expressão no lado direito da atribuição *)
    let (v1,  tdir) = infere_exp amb v
    (* Faz o mesmo para o lado esquerdo *)
    and (ex1, tesq) = infere_exp amb ex in
    (* Os dois tipos devem ser iguais *)
    let _ = mesmo_tipo (posicao v)
                       "Atribuicao com tipos diferentes: %s = %s" tesq
                          tdir in

    let (e1,tinf) = infere_exp amb e in
    (* O tipo inferido para a expressão de teste do comando for deve ser
       booleano *)
    let _ = mesmo_tipo (posicao e)
             "O teste do for deveria ser do tipo %s e nao %s"
             TipoBool tinf in

    (* Infere o tipo da expressão de passo do for *)
```

```
    let (exp1,  tdir) = infere_exp amb exp in

    (* Verifica a validade de cada comando do bloco  *)
    let c1 = List.map (verifica_cmd amb tiporet) c in

     CmdFor ( v1,ex1,e1,exp1,c1)

  | ComandoExpress (e) ->  let (e1,  tinf) = infere_exp amb e in
     ComandoExpress e1

  |CmdCase( v,cas, def) ->
    let (v1,  tinf) = infere_exp amb v in
    let cas1 = List.map (fun elem ->
    match elem with
    Case (exp, cmds) ->
      let (e1,  tinf) = infere_exp amb exp
      in
      let c1 = List.map (verifica_cmd amb tiporet) cmds in
      Case (e1,c1)
    ) cas in
    let def1 = (match def with
      None -> None
    | Some cmds -> Some (List.map (verifica_cmd amb tiporet) cmds)
    ) in
    CmdCase ( v1, cas1, def1)

and verifica_fun amb ast =
  let open A in
  match ast with
    A.Funcao {fn_nome; fn_tiporet; fn_formais; fn_locais; fn_corpo} ->
    (* Estende o ambiente global, adicionando um ambiente local *)
    let ambfn = Amb.novo_escopo amb in
    (* Insere os parâmetros no novo ambiente *)
    let insere_parametro (v,t) = Amb.insere_param ambfn (fst v) t in
    let _ = List.iter insere_parametro fn_formais in
    (* Insere as variáveis locais no novo ambiente *)
    let insere_local = function
        (DecVar (v,t)) -> Amb.insere_local ambfn (fst v)  t in
    let _ = List.iter insere_local fn_locais in
    (* Verifica cada comando presente no corpo da função usando o novo
       ambiente *)
    let corpo_tipado = List.map (verifica_cmd ambfn fn_tiporet) fn_corpo
       in
      A.Funcao {fn_nome; fn_tiporet; fn_formais; fn_locais; fn_corpo =
         corpo_tipado}


let rec verifica_dup xs =
  match xs with
    [] -> []
  | (nome,t)::xs ->
    let id = fst nome in
    if (List.for_all (fun (n,t) -> (fst n) <> id) xs)
    then (id, t) :: verifica_dup xs
    else let msg = "Parametro duplicado " ^ id in
      failwith (msg_erro nome msg)

let insere_declaracao_var amb dec =
  let open A in
```

```
    match dec with
        DecVar (nome, tipo) ->  Amb.insere_local amb (fst nome) tipo

let insere_declaracao_fun amb dec =
  let open A in
    match dec with
      Funcao {fn_nome; fn_tiporet; fn_formais; fn_corpo} ->
        (* Verifica se não há parâmetros duplicados *)
        let formais = verifica_dup fn_formais in
        let nome = fst fn_nome in
        Amb.insere_fun amb nome formais fn_tiporet


(* Lista de cabeçalhos das funções pré definidas *)
let fn_predefs = let open A in [
   ("entrada", [("x", TipoInt); ("y", TipoInt)], TipoVoid);
   ("saida",   [("x", TipoInt); ("y", TipoInt)], TipoVoid)
]

(* insere as funções pré definidas no ambiente global *)
let declara_predefinidas amb =
  List.iter (fun (n,ps,tr) -> Amb.insere_fun amb n ps tr) fn_predefs

let semantico ast =
  (* cria ambiente global inicialmente vazio *)
  let amb_global = Amb.novo_amb [] in
  let _ = declara_predefinidas amb_global in
  let (A.Programa (decs_globais, decs_funs, comandos)) = ast in
  let _ = List.iter (insere_declaracao_var amb_global) decs_globais in
  let _ = List.iter (insere_declaracao_fun amb_global) decs_funs in
  (* Verificação de tipos nas funções *)
  let decs_funs = List.map (verifica_fun amb_global) decs_funs in
  (* Verificação de tipos na função principal *)
  let comandos = List.map (verifica_cmd amb_global A.TipoVoid) comandos in
     (A.Programa (decs_globais, decs_funs, comandos),  amb_global)
```

semanticoTest.ml

```
open Printf
open Lexing

open Ast
exception Erro_Sintatico of string

module S = MenhirLib.General (* Streams *)
module I = Sintatico.MenhirInterpreter

open Semantico

let message =
  fun s ->
    match s with
    | 0 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 1 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 34 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 35 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
```

```
  | 36 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 72 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 47 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 48 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 49 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 51 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 52 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 55 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 56 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 57 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 58 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 61 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 62 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 63 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 64 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 73 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 74 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 95 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 89 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 97 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 98 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 99 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 65 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 66 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 53 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 67 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 68 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 59 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 60 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 42 ->
```

```
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 41 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 70 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 75 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 77 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 76 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 105 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 84 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 43 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 85 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 86 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 45 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 46 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 102 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 103 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 81 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 3 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 2 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 6 ->
    "estado 6: esperava um tipo. Exemplo:\n   x : inteiro;\n"
| 7 ->
    "estado 7: esperava a definicao de um campo. Exemplo:\n   i:
       registro\n         parte_real: inteiro;\n         parte_imag:
       inteiro;\n      fim registro;\n      "
| 8 ->
    "estado 8: esperava ':'. Exemplo:\n   x: inteiro;\n   "
| 9 ->
    "estado 9: esperava um tipo. Exemplo:\n   x: inteiro;\n"
| 25 ->
    "estado 25: esperva um ';'.\n"
| 26 ->
    "estado 26: uma declaracao foi encontrada. Para continuar era\n
       esperado uma outra declara\195\167\195\163o ou a palavra '
       inicio'.\n"
| 29 ->
    "estado 29: espera a palavra 'registro'. Exemplo:\n   i: registro\
       n         parte_real: inteiro;\n         parte_imag: inteiro;\n
             fim registro;\n"
| 31 ->
    "estado 31: esperava um ';'. \n"
| 107 ->
    "estado 107: uma declaracao foi encontrada. Para continuar era\n
```

```
              esperado uma outra declara\195\167\195\163o ou a palavra '
            inicio'.\n"
    | 13 ->
        "estado 13: esperava um '['. Exemplo:\n   arranjo [1..10] de
           inteiro;\n"
    | 14 ->
        "estado 14: esperava os limites do vetor. Exemplo:\n   arranjo
           [1..10] de inteiro;\n"
    | 15 ->
        "estado 15: esperava '..'. Exemplo:\n   1 .. 10\n"
    | 16 ->
        "estado 16: esperava um numero inteiro. Exemplo:\n   1 .. 10\n"
    | 18 ->
        "estado 18: esperava um ']'. Exemplo\n   arranjo [1..10] de
           inteiro;\n"
    | 19 ->
        "estado 19: esperava a palavra reservada 'de'. Exemplo:\n
           arranjo [1..10] de inteiro;\n"
    | 20 ->
        "estado 20: esperava um tipo. Exemplo\n   arranjo [1..10] de
           inteiro;\n"
    | _ ->
        Printf.sprintf "%d" s

let posicao lexbuf =
    let pos = lexbuf.lex_curr_p in
    let lin = pos.pos_lnum
    and col = pos.pos_cnum - pos.pos_bol - 1 in
    sprintf "linha %d, coluna %d" lin col

(* [pilha checkpoint] extrai a pilha do autômato LR(1) contida em
   checkpoint *)

let pilha checkpoint =
  match checkpoint with
  | I.HandlingError amb -> I.stack amb
  | _ -> assert false (* Isso não pode acontecer *)

let estado checkpoint : int =
  match Lazy.force (pilha checkpoint) with
  | S.Nil -> (* O parser está no estado inicial *)
     0
  | S.Cons (I.Element (s, _, _, _), _) ->
     I.number s

let sucesso v = Some v

let falha lexbuf (checkpoint : (Sast.expressao Ast.programa) I.checkpoint)
    =
  let estado_atual = estado checkpoint in
  let msg = message estado_atual in
  raise (Erro_Sintatico (Printf.sprintf "%d - %s.\n"
                                        (Lexing.lexeme_start lexbuf) msg))

let loop lexbuf resultado =
  let fornecedor = I.lexer_lexbuf_to_supplier Lexico.token lexbuf in
  I.loop_handle sucesso (falha lexbuf) fornecedor resultado
```

```
let parse_com_erro lexbuf =
  try
    Some (loop lexbuf (Sintatico.Incremental.programa lexbuf.lex_curr_p))
  with
  | Lexico.Erro msg ->
     printf "Erro lexico na %s:\n\t%s\n" (posicao lexbuf) msg;
     None
  | Erro_Sintatico msg ->
     printf "Erro sintático na %s %s\n" (posicao lexbuf) msg;
     None

let parse s =
  let lexbuf = Lexing.from_string s in
  let ast = parse_com_erro lexbuf in
  ast

let parse_arq nome =
  let ic = open_in nome in
  let lexbuf = Lexing.from_channel ic in
  let ast = parse_com_erro lexbuf in
  let _ = close_in ic in
  ast

let verifica_tipos nome =
  let ast = parse_arq nome in
  match ast with
    Some (Some ast) -> semantico ast
  | _ -> failwith "Nada a fazer!\n"
```

ambiente.ml

```
module Tab = Tabsimb
module A = Ast

type entrada_fn = { tipo_fn:  A.tipo;
                    formais: (string * A.tipo) list;
}

type entrada =  EntFun of entrada_fn
             |  EntVar of A.tipo

type t = {
  ambv : entrada Tab.tabela
}

let novo_amb xs = { ambv = Tab.cria xs }

let novo_escopo amb = { ambv = Tab.novo_escopo amb.ambv }

let busca amb ch = Tab.busca amb.ambv ch

let insere_local amb ch t =
  Tab.insere amb.ambv ch (EntVar t)

let insere_param amb ch t =
  Tab.insere amb.ambv ch (EntVar t)

let insere_fun amb nome params resultado =
  let ef = EntFun { tipo_fn = resultado;
                    formais = params }
```

```
  in Tab.insere amb.ambv nome ef
```

tast.ml

```
open Ast

type expressao = ExpVar of (expressao variavel) * tipo
               | ExpInt of int * tipo
               | ExpString of string * tipo
               | ExpChar of char * tipo
               | ExpReal of float * tipo
               | ExpBool of bool * tipo
               | ExpOp of (oper * tipo) * (expressao * tipo) * (expressao *
                  tipo)
               | ExpChamada of ident * (expressao expressoes) * tipo
```

tabsimb.ml

```
type 'a tabela = {
    tbl: (string, 'a) Hashtbl.t;
    pai: 'a tabela option;
}

exception Entrada_existente of string;;

let insere amb ch v =
  if Hashtbl.mem amb.tbl ch
  then raise (Entrada_existente ch)
  else Hashtbl.add amb.tbl ch v

let substitui amb ch v = Hashtbl.replace amb.tbl ch v

let rec atualiza amb ch v =
    if Hashtbl.mem amb.tbl ch
    then Hashtbl.replace amb.tbl ch v
    else match amb.pai with
       None -> failwith "tabsim atualiza: chave nao encontrada"
     | Some a -> atualiza a ch v

let rec busca amb ch =
  try Hashtbl.find amb.tbl ch
  with Not_found ->
    (match amb.pai with
       None -> raise Not_found
     | Some a -> busca a ch)

let rec cria cvs =
  let amb = {
    tbl = Hashtbl.create 5;
    pai = None
  } in
  let _ = List.iter (fun (c,v) -> insere amb c v) cvs
  in amb

let novo_escopo amb_pai = {
  tbl = Hashtbl.create 5;
  pai = Some amb_pai
}
```

6.2 Compilação do analisador semântico

Para compilar o analisador semântico implementado de acordo com os arquivos mostrados na seção anterior, será utilizado o Menhir e o arquivo "semanticoTest". A compilação pode ser feita através do sguinte comando:

```
ocamlbuild -use-ocamlfind -use-menhir -menhir "menhir --table" -package
    menhirLib semanticoTest.byte
```

6.2.1 Para facilitar:

Usamos um script para compilar todo o projeto.
script.sh

```
#!/bin/bash

rm -rf  _build  semanticoTest.byte

menhir -v --list-errors sintatico.mly --compile-errors sintatico.messages
    > fnmes.ml

ocamlbuild -use-ocamlfind -use-menhir -menhir "menhir --table" -package
    menhirLib semanticoTest.byte

rlwrap ocaml

# para dar permissao pro script rodar digita no terminal chmod 777 script.
    sh
#para executar bash script.sh
```

6.2.2 Teste do analisador semântico

Para testar o analisador semântico será usado um arquivo com o programa fonte escrito em JavaScript.
Uma vez dentro do ocaml, o teste é feito a partir dos seguintes comandos:

```
#se desejar ver apenas a árvore sintática que sai do analisador sintático,
     digite
    parse_arq "arquivo.js";;
#Depois, para ver a saída do analisador semântico já com a árvore anotada
    com o tipos, digite:
    verifica_tipos "arquivo.js";;
#Note que o analisador semântico está retornando também o ambiente global.
```

A árvore tipada gerada pelo analisador semântico para o programa fonte apresentado é mostrada a seguir.
Árvore tipada:

```
# verifica_tipos "micro10.js";;
- : Tast.expressao Ast.programa * Ambiente.t =
(Programa
  ([DecVar
     (("numero",
       {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 4}),
      TipoInt);
    DecVar
     (("fat",
```

```
     {Lexing.pos_fname = ""; pos_lnum = 2; pos_bol = 21; pos_cnum = 25})
        ,
   TipoInt)],
[Funcao
  {Ast.fn_nome =
    ("fatorial",
     {Lexing.pos_fname = ""; pos_lnum = 5; pos_bol = 41; pos_cnum = 50})
       ;
   fn_tiporet = TipoInt;
   fn_formais =
    [(("numero",
       {Lexing.pos_fname = ""; pos_lnum = 5; pos_bol = 41; pos_cnum =
          59}),
      TipoInt)];
   fn_locais = [];
   fn_corpo =
    [CmdSe
      (Tast.ExpOp ((Menor, TipoBool),
        (Tast.ExpVar
          (VarSimples
            ("numero",
             {Lexing.pos_fname = ""; pos_lnum = 6; pos_bol = 88;
              pos_cnum = 96}),
         TipoInt),
        TipoInt),
        (Tast.ExpInt (0, TipoInt), TipoInt)),
      [CmdRetorno (Some (Tast.ExpInt (1, TipoInt)))],
      Some
       [CmdRetorno
         (Some
           (Tast.ExpOp ((Mult, TipoInt),
             (Tast.ExpVar
               (VarSimples
                 ("numero",
                  {Lexing.pos_fname = ""; pos_lnum = 9; pos_bol = 138;
                   pos_cnum = 153}),
              TipoInt),
             TipoInt),
             (Tast.ExpChamada ("fatorial",
               [Tast.ExpOp ((Menos, TipoInt),
                 (Tast.ExpVar
                   (VarSimples
                     ("numero",
                      {Lexing.pos_fname = ""; pos_lnum = 9; pos_bol =
                         138;
                       pos_cnum = 171}),
                    TipoInt),
                   TipoInt),
                 (Tast.ExpInt (1, TipoInt), TipoInt))],
              TipoInt),
             TipoInt))))])]}],
[CmdSaida [Tast.ExpString ("Digite um numero", TipoString)];
 CmdEntrada
  (Tast.ExpVar
    (VarSimples
      ("numero",
       {Lexing.pos_fname = ""; pos_lnum = 14; pos_bol = 226;
        pos_cnum = 235}),
    TipoInt));
```

```
    CmdAtrib
     (Tast.ExpVar
       (VarSimples
         ("fat",
          {Lexing.pos_fname = ""; pos_lnum = 15; pos_bol = 244;
           pos_cnum = 244}),
       TipoInt),
     Tast.ExpChamada ("fatorial",
      [Tast.ExpVar
        (VarSimples
          ("numero",
           {Lexing.pos_fname = ""; pos_lnum = 15; pos_bol = 244;
            pos_cnum = 259}),
        TipoInt)],
      TipoInt));
    CmdSaida
     [Tast.ExpString ("O fatorial de: ", TipoString);
      Tast.ExpVar
       (VarSimples
         ("numero",
          {Lexing.pos_fname = ""; pos_lnum = 16; pos_bol = 268;
           pos_cnum = 298}),
       TipoInt);
      Tast.ExpString (" \195\169", TipoString);
      Tast.ExpVar
       (VarSimples
         ("fat",
          {Lexing.pos_fname = ""; pos_lnum = 16; pos_bol = 268;
           pos_cnum = 311}),
       TipoInt)]]),
  <abstr>)
```

Caso tenha algum erro, como por exemplo função não existente:

```
# verifica_tipos "micro10.js";;
Exception:
Failure
 "Semantico -> linha 9, coluna 23: Nao existe a funcao de nome fatoria".
```

7 Interpretador

Essa etapa faz a avaliação das expressões sem se preocupar com qualquer tipo de erro, pois esses ja foram tratados nas etapas anteriores. Há a preocupação com variaveis não inicializadas ou rótulos de variáveis já usadas, além de tipos digitados errados pelo usuário.

Passos para construir o Interprete a partir do Semantico

```
1 - Copiar pasta do semântico para uma nova pasta.
2 - Copiar os arquivos semantico.ml, semantico.mli e semanticoTest.ml para
     interprete.ml, interprete.mli e interpreteTest.ml, respectivamente.
3 - Copiar os arquivos ambiente.ml e ambiente.mli para ambInterp.ml e
    ambInterp.mli.

Visualize a árvore resultante do semântico para algum arquivo de interesse
    .

- Em ambInterp.ml:
```

```
  - Alterar 'entrada' para envolver valores
  - Inserir 'atualiza_var'

Em interprete.ml:
  - renomear a fç semantico para interprete
  - alterar fn_predefs
  - alterar 'declara_predefinidas'
  - alterar 'insere_declaracao_var'
  - renomear todas as ocorrências de 'verifica_dup' para 'obtem_formais' e
      alterar o corpo;
  - alterar 'insere_declaracao_fun'
  - alterar 'interprete', removendo 'let decs_funs'
  - alterar 'tast.ml' para incluir 'ExpVoid'
  - renomear todas as ocorrências de 'verifica_cmd' para 'interpreta_cmd'
     e altere o corpo
  - renomear todas as ocorrências de 'infere_exp' para 'interpreta_exp'
     renomear todas as ocorrências
  - Inserir 'obtem_nome_var'
  - alterar ''
  - Remover 'posicao', 'msg_erro_pos', 'msg_erro' e 'nome_tipo'
```

7.1 Arquivos do Intérprete

Nessa seção será mostrado apenas os arquivos alterados para a construção do intérprete.

7.1.1 Intérprete

interprete.ml

```
module Amb = AmbInterp
module A = Ast
module S = Sast
module T = Tast

exception Valor_de_retorno of T.expressao

let obtem_nome_tipo_var exp = let open T in
  match exp with
  | ExpVar (v,tipo) ->
    (match v with
      | A.VarSimples (nome,_) -> (nome,tipo)
    )
  | _ -> failwith "obtem_nome_tipo_var: nao eh variavel"

let pega_int exp =
  match exp with
  |  T.ExpInt (i,_) -> i
  | _ -> failwith "pega_int: nao eh inteiro"

let pega_float exp =
  match exp with
  |  T.ExpReal (i,_) -> i
  | _ -> failwith "pega_float: nao eh float"

let pega_string exp =
  match exp with
  |  T.ExpString (s,_) -> s
  | _ -> failwith "pega_string: nao eh string"
```

```
let pega_char exp =
  match exp with
  |  T.ExpChar (i,_) -> i
  | _ -> failwith "pega_char: nao eh caracter"

let pega_bool exp =
  match exp with
  |  T.ExpBool (b,_) -> b
  | _ -> failwith "pega_bool: nao eh booleano"

type classe_op = Aritmetico | Relacional | Logico | Cadeia

let classifica op =
  let open A in
  match op with
   And
  | Or -> Logico
  | Mais
  | Menos
  | Mult
  | Div
  | Mod
  | Expoente -> Aritmetico
  | Menor
  | Maior
  | MenorIgual
  | Igual
  | MaiorIgual
  | Difer -> Relacional
  | Concat -> Cadeia


let rec interpreta_exp amb exp =
let open A in
let open T in

  match exp with
  | ExpVoid
  | ExpInt _
  | ExpString _
  | ExpChar _
  | ExpReal _
  | ExpBool _   -> exp
  | ExpVar _ ->
    let (id,tipo) = obtem_nome_tipo_var exp in
    (* Tenta encontrar o valor da variável no escopo local, se não      *)
    (* encontrar, tenta novamente no escopo que engloba o atual. Prossegue
       -se *)
    (* assim até encontrar o valor em algum escopo englobante ou até    *)
    (* encontrar o escopo global. Se em algum lugar for encontrado,
              *)
    (* devolve-se o valor. Em caso contrário, devolve uma exceção       *)
    (match (Amb.busca amb id) with
     | Amb.EntVar (tipo, v) ->
       (match v with
        | None -> failwith ("variável nao inicializada: " ^ id)
        | Some valor -> valor
       )
```

```
    |  _ -> failwith "interpreta_exp: expvar"
  )
| ExpOp ((op,top), (esq, tesq), (dir,tdir)) ->
  let  vesq = interpreta_exp amb esq
  and vdir = interpreta_exp amb dir in

  let interpreta_aritmetico () =
    (match tesq with
     | TipoInt ->
       (match op with
        | Mais  -> ExpInt (pega_int vesq + pega_int vdir, top)
        | Menos -> ExpInt (pega_int vesq - pega_int vdir, top)
        | Mult  -> ExpInt (pega_int vesq * pega_int vdir, top)
        | Div   -> ExpInt (pega_int vesq / pega_int vdir, top)
        | _ -> failwith "interpreta_aritmetico"
       )
     | TipoReal ->
       (match op with
        | Mais  -> ExpReal (pega_float vesq +. pega_float vdir, top)
        | Menos -> ExpReal (pega_float vesq -. pega_float vdir, top)
        | Mult  -> ExpReal (pega_float vesq *. pega_float vdir, top)
        | Div   -> ExpReal (pega_float vesq /. pega_float vdir, top)
        | _ -> failwith "interpreta_aritmetico"
       )
     | _ -> failwith "interpreta_aritmetico"
    )

  and interpreta_relacional () =
    (match tesq with
     | TipoInt ->
       (match op with
        | Menor -> ExpBool (pega_int vesq < pega_int vdir, top)
        | MenorIgual -> ExpBool (pega_int vesq <= pega_int vdir, top)
        | Maior  -> ExpBool (pega_int vesq > pega_int vdir, top)
        | MaiorIgual  -> ExpBool (pega_int vesq >= pega_int vdir, top)
        | Igual   -> ExpBool (pega_int vesq == pega_int vdir, top)
        | Difer   -> ExpBool (pega_int vesq != pega_int vdir, top)
        | _ -> failwith "interpreta_relacional int"
       )
     | TipoReal ->
       (match op with
        | Menor -> ExpBool (pega_float vesq < pega_float vdir, top)
        | MenorIgual -> ExpBool (pega_float vesq <= pega_float vdir, top
           )
        | Maior  -> ExpBool (pega_float vesq > pega_float vdir, top)
        | MaiorIgual  -> ExpBool (pega_float vesq >= pega_float vdir,
          top)
        | Igual   -> ExpBool (pega_float vesq == pega_float vdir, top)
        | Difer   -> ExpBool (pega_float vesq != pega_float vdir, top)
        | _ -> failwith "interpreta_relacional f"
       )
     | TipoString ->
       (match op with
        | Menor -> ExpBool (pega_string vesq < pega_string vdir, top)
        | Maior  -> ExpBool (pega_string vesq > pega_string vdir, top)
        | Igual   -> ExpBool (pega_string vesq == pega_string vdir, top)
        | Difer   -> ExpBool (pega_string vesq != pega_string vdir, top)
        | _ -> failwith "interpreta_relacional"
       )
```

```
        | TipoBool ->
          (match op with
           | Menor -> ExpBool (pega_bool vesq < pega_bool vdir, top)
           | Maior  -> ExpBool (pega_bool vesq > pega_bool vdir, top)
           | Igual   -> ExpBool (pega_bool vesq == pega_bool vdir, top)
           | Difer   -> ExpBool (pega_bool vesq != pega_bool vdir, top)
           | _ -> failwith "interpreta_relacional"
          )
        | _ ->  failwith "interpreta_relacional"
       )

    and interpreta_logico () =
      (match tesq with
       | TipoBool ->
          (match op with
           | Or -> ExpBool (pega_bool vesq || pega_bool vdir, top)
           | And ->   ExpBool (pega_bool vesq && pega_bool vdir, top)
           | _ ->  failwith "interpreta_logico"
          )
       | _ ->  failwith "interpreta_logico"
      )
    and interpreta_cadeia () =
      (match tesq with
       | TipoString ->
          (match op with
           | Concat -> ExpString (pega_string vesq ^ pega_string vdir, top)
           | _ ->  failwith "interpreta_cadeia"
          )
       | _ ->  failwith "interpreta_cadeia"
      )
      in
    let valor = (match (classifica op) with
          Aritmetico -> interpreta_aritmetico ()
        | Logico -> interpreta_logico ()
        | Relacional -> interpreta_relacional ()
        | Cadeia -> interpreta_cadeia ()
      )
    in
      valor


  | ExpChamada (id, args, tipo) ->
    let open Amb in
    ( match (Amb.busca amb id) with
      | Amb.EntFun {tipo_fn; formais; locais; corpo} ->
           (* Interpreta cada um dos argumentos *)
           let vargs = List.map (interpreta_exp amb) args in
           (* Associa os argumentos aos parâmetros formais *)
           let vformais = List.map2 (fun (n,t) v -> (n, t, Some v))
               formais vargs
           in interpreta_fun amb id vformais locais corpo
      | _ -> failwith "interpreta_exp: expchamada"
    )

and interpreta_fun amb fn_nome fn_formais fn_locais fn_corpo =
  let open A in
 (* Estende o ambiente global, adicionando um ambiente local *)
  let ambfn = Amb.novo_escopo amb in
   let insere_local  d =
```

```
    match d with
      (DecVar (v,t)) -> Amb.insere_local ambfn (fst v)  t None
  in
  (* Associa os argumentos aos parâmetros e insere no novo ambiente *)
  let insere_parametro (n,t,v) = Amb.insere_param ambfn n t v in
  let _ = List.iter insere_parametro fn_formais in
  (* Insere as variáveis locais no novo ambiente *)
    let _ = List.iter insere_local fn_locais in
    (* Interpreta cada comando presente no corpo da função usando o novo
       ambiente *)
  try
    let _ = List.iter (interpreta_cmd ambfn) fn_corpo in T.ExpVoid
    with
       Valor_de_retorno expret -> expret

and interpreta_cmd amb cmd =
  let open A in
  let open T in
  match cmd with
    CmdRetorno exp ->
    (* Levantar uma exceção foi necessária pois, pela semântica do comando
        de
       retorno, sempre que ele for encontrado em uma função, a computação
       deve parar retornando o valor indicado, sem realizar os demais
          comandos.
    *)
    (match exp with
     (* Se a função não retornar nada, verifica se ela foi declarada como
        void *)
      None -> raise (Valor_de_retorno ExpVoid)
     | Some e ->
       (* Avalia a expressão e retorne o resultado *)
       let e1 = interpreta_exp amb e in
       raise (Valor_de_retorno e1)
    )
  | CmdSe (teste, entao, senao) ->
    let teste1 = interpreta_exp amb teste in
   (match teste1 with
       ExpBool (true,_) ->
       (* Interpreta cada comando do bloco 'então' *)
       List.iter (interpreta_cmd amb) entao
     | _ ->
       (* Interpreta cada comando do bloco 'senão', se houver *)
       (match senao with
          None -> ()
        | Some bloco -> List.iter (interpreta_cmd amb) bloco
       )
    )

  | CmdAtrib (elem, exp) ->
    (* Interpreta o lado direito da atribuição *)
    let exp = interpreta_exp amb exp
    (* Faz o mesmo para o lado esquerdo *)
    and (elem1,tipo) = obtem_nome_tipo_var elem in
    Amb.atualiza_var amb elem1 tipo (Some exp)

  | ComandoExpress exp -> ignore( interpreta_exp amb exp)

  | CmdEntrada exp ->
```

```
    (* Obtem os nomes e os tipos de cada um dos argumentos *)
    let nts = obtem_nome_tipo_var exp in
    let leia_var (nome,tipo) =
      let valor =
        (match tipo with
         | A.TipoInt    -> T.ExpInt    (read_int (),   tipo)
         | A.TipoString -> T.ExpString (read_line (),  tipo)
         | A.TipoReal   -> T.ExpReal   (read_float (), tipo)
         | A.TipoChar   ->let str = (read_line ()).[0] in T.ExpChar   (str
            ,  tipo)
         | _ -> failwith "leia_var: nao implementado"
        )
      in  Amb.atualiza_var amb nome tipo (Some valor)
    in
    (* Lê o valor para cada argumento e atualiza o ambiente *)
    leia_var nts

  | CmdEntrada exps ->
    let _ = interpreta_cmd amb (CmdEntrada exps) in
      print_newline()

  | CmdSaida exps ->
    (* Interpreta cada argumento da função 'saida' *)
    let exps = List.map (interpreta_exp amb) exps in
    let imprima exp =
      (match exp with
       | T.ExpInt (n,_) ->      let _ = print_int n in print_string " "
       | T.ExpString (s,_) -> let _ = print_string s in print_string " "
       | T.ExpBool (b,_) ->
         let _ = print_string (if b then "true" else "false")
         in print_string " "
       | _ -> failwith "imprima: nao implementado"
      )
    in
    let _ = List.iter imprima exps in
    print_newline ()

  | CmdSaida exps ->
    (* Interpreta cada argumento da função 'saida' *)
    let exps = List.map (interpreta_exp amb) exps in
    let imprima exp =
      (match exp with
       | T.ExpInt (n,_) ->      let _ = print_int n in print_string " "
       | T.ExpString (s,_) -> let _ = print_string s in print_string " "
       | T.ExpBool (b,_) ->
         let _ = print_string (if b then "true" else "false")
         in print_string " "
       | _ -> failwith "imprima: nao implementado"
      )
    in
    let _ = List.iter imprima exps in
    let _ = print_newline () in
      print_newline ()


  | CmdWhile (teste, doit) ->
     let teste1 = interpreta_exp amb teste in
   (match teste1 with
      ExpBool (true,_) ->
```

```
        (* Interpreta uma iteraçao comando do corpo do while *)
        let _ = List.iter (interpreta_cmd amb) doit in
        (* interpreta recursivamente as possíveis demais iterações do
           comando *)
         interpreta_cmd amb (CmdWhile (teste, doit))
        | _ -> ()
    )

  | CmdCase (teste, cases, senao) ->
     let teste1 = interpreta_exp amb teste in
    let rec match_cases listacase = ( match listacase with
      head::tail ->
      (* Percorre a lista de cases *)
      (match head with Case (l,c) ->
      (* se a expressao do case for igual
      a expressao do comando swith, então*)
       if (interpreta_exp amb l) = teste1 then
        (* avalie cada comando desse case *)
        List.iter (interpreta_cmd amb) c
        (*caso não sejam iguais, avalie o próximo case*)
       else match_cases tail )
        (* se alcançou o fim da lista, é porque não acho nenhum case
           compatível,
        logo, hora de executar o bloco default*)
      | [] ->( match senao with
        Some c ->
        List.iter (interpreta_cmd amb) c
        (* não sei de um comando return em Ocaml,
         acredito que esse seja o jeito correto*)
        | None -> ignore()) )
    in match_cases cases

    | CmdFor (variavel, exp, condicao, var,inc, doit) ->
    let _ = interpreta_cmd amb (CmdAtrib (variavel, exp)) in
    (* let inc = interpreta_cmd amb (CmdAtrib (var, inc)) in *)
    let inc = CmdAtrib (var, inc) in
    interpreta_cmd amb (CmdWhile (condicao, List.append doit [inc]))


let insere_declaracao_var amb dec =
    match dec with
        A.DecVar (nome, tipo) ->  Amb.insere_local amb (fst nome) tipo
           None

let insere_declaracao_fun amb dec =
  let open A in
    match dec with
      Funcao {fn_nome; fn_tiporet; fn_formais; fn_locais; fn_corpo} ->
        let nome = fst fn_nome in
        let formais = List.map (fun (n,t) -> ((fst n), t)) fn_formais in
        Amb.insere_fun amb nome formais fn_locais fn_tiporet fn_corpo


(* Lista de cabeçalhos das funções pré definidas *)
let fn_predefs = let open A in [
    ("entrada", [("x", TipoInt); ("y", TipoInt)], TipoVoid, []);
    ("saida",     [("x", TipoInt); ("y", TipoInt)], TipoVoid, []);
]
```

```
(* insere as funções pré definidas no ambiente global *)
let declara_predefinidas amb =
  List.iter (fun (n,ps,tr,c) -> Amb.insere_fun amb n ps [] tr c)
     fn_predefs

let interprete ast =
  (* cria ambiente global inicialmente vazio *)
  let amb_global = Amb.novo_amb [] in
  let _ = declara_predefinidas amb_global in
  let (A.Programa ( decs_globais, decs_funs, corpo)) = ast in
  let _ = List.iter (insere_declaracao_var amb_global) decs_globais in
  let _ = List.iter (insere_declaracao_fun amb_global) decs_funs in
  (* Interpreta a função principal *)
  let resultado = List.iter (interpreta_cmd amb_global) corpo in
  resultado
```

interpreteTest.ml

```
open Printf
open Lexing

open Ast
exception Erro_Sintatico of string

module S = MenhirLib.General (* Streams *)
module I = Sintatico.MenhirInterpreter

open Semantico

let message =
  fun s ->
    match s with
    | 0 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 1 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 34 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 35 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 36 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 72 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 47 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 48 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 49 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 51 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 52 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 55 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 56 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 57 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
```

```
  | 58 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 61 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 62 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 63 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 64 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 73 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 74 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 95 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 89 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 97 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 98 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 99 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 65 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 66 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 53 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 67 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 68 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 59 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 60 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 42 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 41 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 70 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 75 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 77 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 76 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 105 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 84 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 43 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 85 ->
      "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
  | 86 ->
```

```
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 45 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 46 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 102 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 103 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 81 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 3 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 2 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 6 ->
    "estado 6: esperava um tipo. Exemplo:\n   x : inteiro;\n"
| 7 ->
    "estado 7: esperava a definicao de um campo. Exemplo:\n   i:
       registro\n         parte_real: inteiro;\n         parte_imag:
       inteiro;\n      fim registro;\n      "
| 8 ->
    "estado 8: esperava ':'. Exemplo:\n   x: inteiro;\n   "
| 9 ->
    "estado 9: esperava um tipo. Exemplo:\n   x: inteiro;\n"
| 25 ->
    "estado 25: esperva um ';'.\n"
| 26 ->
    "estado 26: uma declaracao foi encontrada. Para continuar era\n
       esperado uma outra declara\195\167\195\163o ou a palavra '
       inicio'.\n"
| 29 ->
    "estado 29: espera a palavra 'registro'. Exemplo:\n   i: registro\
       n         parte_real: inteiro;\n         parte_imag: inteiro;\n
             fim registro;\n"
| 31 ->
    "estado 31: esperava um ';'. \n"
| 107 ->
    "estado 107: uma declaracao foi encontrada. Para continuar era\n
         esperado uma outra declara\195\167\195\163o ou a palavra '
       inicio'.\n"
| 13 ->
    "estado 13: esperava um '['. Exemplo:\n   arranjo [1..10] de
       inteiro;\n"
| 14 ->
    "estado 14: esperava os limites do vetor. Exemplo:\n   arranjo
       [1..10] de inteiro;\n"
| 15 ->
    "estado 15: esperava '..'. Exemplo:\n   1 .. 10\n"
| 16 ->
    "estado 16: esperava um numero inteiro. Exemplo:\n   1 .. 10\n"
| 18 ->
    "estado 18: esperava um ']'. Exemplo\n   arranjo [1..10] de
       inteiro;\n"
| 19 ->
    "estado 19: esperava a palavra reservada 'de'. Exemplo:\n
       arranjo [1..10] de inteiro;\n"
| 20 ->
    "estado 20: esperava um tipo. Exemplo\n   arranjo [1..10] de
```

```
            inteiro;\n"
    | _ ->
        raise Not_found

let posicao lexbuf =
    let pos = lexbuf.lex_curr_p in
    let lin = pos.pos_lnum
    and col = pos.pos_cnum - pos.pos_bol - 1 in
    sprintf "linha %d, coluna %d" lin col

(* [pilha checkpoint] extrai a pilha do autômato LR(1) contida em
   checkpoint *)

let pilha checkpoint =
  match checkpoint with
  | I.HandlingError amb -> I.stack amb
  | _ -> assert false (* Isso não pode acontecer *)

let estado checkpoint : int =
  match Lazy.force (pilha checkpoint) with
  | S.Nil -> (* O parser está no estado inicial *)
     0
  | S.Cons (I.Element (s, _, _, _), _) ->
     I.number s

let sucesso v = Some v

let falha lexbuf (checkpoint : (Sast.expressao Ast.programa) I.checkpoint)
    =
  let estado_atual = estado checkpoint in
  let msg = message estado_atual in
  raise (Erro_Sintatico (Printf.sprintf "%d - %s.\n"
                                        (Lexing.lexeme_start lexbuf) msg))

let loop lexbuf resultado =
  let fornecedor = I.lexer_lexbuf_to_supplier Lexico.token lexbuf in
  I.loop_handle sucesso (falha lexbuf) fornecedor resultado


let parse_com_erro lexbuf =
  try
    Some (loop lexbuf (Sintatico.Incremental.programa lexbuf.lex_curr_p))
  with
  | Lexico.Erro msg ->
     printf "Erro lexico na %s:\n\t%s\n" (posicao lexbuf) msg;
     None
  | Erro_Sintatico msg ->
     printf "Erro sintático na %s %s\n" (posicao lexbuf) msg;
     None

let parse s =
  let lexbuf = Lexing.from_string s in
  let ast = parse_com_erro lexbuf in
  ast

let parse_arq nome =
  let ic = open_in nome in
  let lexbuf = Lexing.from_channel ic in
  let ast = parse_com_erro lexbuf in
```

```
  let _ = close_in ic in
  ast

let verifica_tipos nome =
  let ast = parse_arq nome in
  match ast with
    Some (Some ast) -> semantico ast
  | _ -> failwith "Nada a fazer!\n"


let interprete nome =
  let tast,amb = verifica_tipos nome in
  Interprete.interprete tast
```

7.1.2 Ambiente Intérprete

ambInterp.ml

```
module Tab = Tabsimb
module A = Ast
module T = Tast

type entrada_fn = {
  tipo_fn:  A.tipo;
  formais: (string * A.tipo) list;
  locais:  A.declaracoes;
  corpo: T.expressao A.comandos
}

type entrada =  EntFun of entrada_fn
             |  EntVar of A.tipo * (T.expressao option)

type t = {
  ambv : entrada Tab.tabela
}

let novo_amb xs = { ambv = Tab.cria xs }

let novo_escopo amb = { ambv = Tab.novo_escopo amb.ambv }

let busca amb ch = Tab.busca amb.ambv ch

let atualiza_var amb ch t v =
  Tab.atualiza amb.ambv ch (EntVar (t,v))

let insere_local amb nome t v =
  Tab.insere amb.ambv nome (EntVar (t,v))

let insere_param amb nome t v =
  Tab.insere amb.ambv nome (EntVar (t,v))

let insere_fun amb nome params locais resultado corpo =
  let ef = EntFun { tipo_fn = resultado;
                    formais = params;
                    locais = locais;
                    corpo = corpo }
  in Tab.insere amb.ambv nome ef
```

7.1.3 Tast

tast.ml

```
open Ast

type expressao = ExpVar of (expressao variavel) * tipo
               | ExpInt of int * tipo
               | ExpString of string * tipo
               | ExpChar of char * tipo
               | ExpReal of float * tipo
               | ExpVoid
               | ExpBool of bool * tipo
               | ExpOp of (oper * tipo) * (expressao * tipo) * (expressao *
                   tipo)
               | ExpChamada of ident * (expressao expressoes) * tipo
```

7.2 Compilação do Intérprete

.ocamlinit

```
let () =
  try Topdirs.dir_directory (Sys.getenv "OCAML_TOPLEVEL_PATH")
  with Not_found -> ()
;;

#use "topfind";;
#require "menhirLib";;
#directory "_build";;
#load "sintatico.cmo";;
#load "lexico.cmo";;
#load "ast.cmo";;
#load "sast.cmo";;
#load "tast.cmo";;
#load "tabsimb.cmo";;
#load "ambiente.cmo";;
#load "semantico.cmo";;
#load "ambInterp.cmo";;
#load "interprete.cmo";;
#load "interpreteTest.cmo";;

open Ast
open Ambiente
open InterpreteTest
```

Comandos no terminal

```
rm -rf  _build  interpreteTest.byte
## opcional: funciona da mesma forma sem esse comando

ocamlbuild -use-ocamlfind -use-menhir -menhir "menhir --table" -package
    menhirLib interpreteTeste.byte

rlwrap ocaml

interprete "arquivo.js";;
```

Observação: após alguns testes foi notado que o uso de um script com os comandos acima da erro no ocamlinit, e se executar os comandos individualmente no terminal o mesmo erro não ocorre.

7.3 Testes Feitos e Resultados Obtidos

Exemplos com respostas de erro:

```
# interprete "micro09.js";;
Exception:
Failure
 "Semantico -> linha 11, coluna 12: um operador relacional nao pode ser
    usado com o tipo float".


# interprete "micro8.js";;
Erro sintático na linha 3, coluna 10 22 - 17.

Exception: Failure "Nada a fazer!\n".
```

Exemplos com execução sem erros:

Teste micro08

```
# interprete "micro08.js";;
Digite um numero:
8
O numero 8 eh menor que 10
Digite um numero:
11
O numero 11 eh maior que 10
Digite um numero:
0
O numero 0 eh menor que 10
- : unit = ()
```

Teste micro10

```
# interprete "micro10.js";;
Digite um numero
6
O fatorial de:  6  é 720
- : unit = ()
```

Teste micro11

```
# interprete "micro11.js";;
Digite um número
3
Numero positivo
- : unit = ()
# interprete "micro11.js";;
Digite um número
-3
Numero negativo
- : unit = ()
# interprete "micro11.js";;
Digite um número
0
Zero
- : unit = ()
```

7.3.1 Curiosidade

As operações das seções anteriores também podem ser realizadas, como por exemplo: obter a árvore sintática abstrata ou a árvore tipada.

```
# parse_arq "micro10.js";;
- : Sast.expressao Ast.programa option option =
Some
 (Some
   (Programa
     ([DecVar
        (("numero",
          {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum =
             4}),
         TipoInt);
       DecVar
        (("fat",
          {Lexing.pos_fname = ""; pos_lnum = 2; pos_bol = 21; pos_cnum =
             25}),
         TipoInt)],
     [Funcao
       {fn_nome =
         ("fatorial",
          {Lexing.pos_fname = ""; pos_lnum = 5; pos_bol = 41; pos_cnum =
             50});
        fn_tiporet = TipoInt;
        fn_formais =
         [(("numero",
            {Lexing.pos_fname = ""; pos_lnum = 5; pos_bol = 41;
             pos_cnum = 59}),
           TipoInt)];
        fn_locais = [];
        fn_corpo =
         [CmdSe
           (Sast.ExpOp
             ((MenorIgual,
               {Lexing.pos_fname = ""; pos_lnum = 6; pos_bol = 88;
                pos_cnum = 103}),
              Sast.ExpVar
               (VarSimples
                 ("numero",
                  {Lexing.pos_fname = ""; pos_lnum = 6; pos_bol = 88;
                   pos_cnum = 96})),
              Sast.ExpInt
               (0,
                {Lexing.pos_fname = ""; pos_lnum = 6; pos_bol = 88;
                 pos_cnum = 106})),
            [CmdRetorno
              (Some
                (Sast.ExpInt
                  (1,
                   {Lexing.pos_fname = ""; pos_lnum = 7; pos_bol = 110;
                    pos_cnum = 125})))],
            Some
             [CmdRetorno
               (Some
                 (Sast.ExpOp
                   ((Mult,
                     {Lexing.pos_fname = ""; pos_lnum = 9; pos_bol = 139;
                      pos_cnum = 161}),
```

```
                Sast.ExpVar
                 (VarSimples
                   ("numero",
                    {Lexing.pos_fname = ""; pos_lnum = 9; pos_bol = 139;
                     pos_cnum = 154})),
                Sast.ExpChamada
                 (("fatorial",
                   {Lexing.pos_fname = ""; pos_lnum = 9; pos_bol = 139;
                    pos_cnum = 163}),
                  [Sast.ExpOp
                    ((Menos,
                      {Lexing.pos_fname = ""; pos_lnum = 9; pos_bol =
                         139;
                       pos_cnum = 179}),
                     Sast.ExpVar
                      (VarSimples
                        ("numero",
                         {Lexing.pos_fname = ""; pos_lnum = 9; pos_bol =
                            139;
                          pos_cnum = 172})),
                     Sast.ExpInt
                      (1,
                       {Lexing.pos_fname = ""; pos_lnum = 9; pos_bol =
                          139;
                        pos_cnum = 181}))])))])]}],
[CmdSaida
  [Sast.ExpString
    ("Digite um numero",
     {Lexing.pos_fname = ""; pos_lnum = 13; pos_bol = 194;
      pos_cnum = 223})];
 CmdEntrada
  [Sast.ExpVar
    (VarSimples
      ("numero",
       {Lexing.pos_fname = ""; pos_lnum = 14; pos_bol = 227;
        pos_cnum = 236}))];
 CmdAtrib
  (Sast.ExpVar
    (VarSimples
      ("fat",
       {Lexing.pos_fname = ""; pos_lnum = 15; pos_bol = 245;
        pos_cnum = 245})),
   Sast.ExpChamada
    (("fatorial",
      {Lexing.pos_fname = ""; pos_lnum = 15; pos_bol = 245;
       pos_cnum = 251}),
     [Sast.ExpVar
       (VarSimples
         ("numero",
          {Lexing.pos_fname = ""; pos_lnum = 15; pos_bol = 245;
           pos_cnum = 260}))]));
 CmdSaida
  [Sast.ExpString
    ("O fatorial de: ",
     {Lexing.pos_fname = ""; pos_lnum = 16; pos_bol = 269;
      pos_cnum = 297});
   Sast.ExpVar
    (VarSimples
      ("numero",
```

```
            {Lexing.pos_fname = ""; pos_lnum = 16; pos_bol = 269;
             pos_cnum = 299}));
        Sast.ExpString
         (" \195\169",
          {Lexing.pos_fname = ""; pos_lnum = 16; pos_bol = 269;
           pos_cnum = 310});
        Sast.ExpVar
         (VarSimples
           ("fat",
            {Lexing.pos_fname = ""; pos_lnum = 16; pos_bol = 269;
             pos_cnum = 312}))]])))


# verifica_tipos "micro10.js";;
- : Tast.expressao Ast.programa * Ambiente.t =
(Programa
  ([DecVar
     (("numero",
       {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 4}),
      TipoInt);
    DecVar
     (("fat",
       {Lexing.pos_fname = ""; pos_lnum = 2; pos_bol = 21; pos_cnum = 25})
         ,
      TipoInt)],
  [Funcao
    {fn_nome =
      ("fatorial",
       {Lexing.pos_fname = ""; pos_lnum = 5; pos_bol = 41; pos_cnum = 50})
         ;
     fn_tiporet = TipoInt;
     fn_formais =
      [(("numero",
         {Lexing.pos_fname = ""; pos_lnum = 5; pos_bol = 41; pos_cnum =
            59}),
        TipoInt)];
     fn_locais = [];
     fn_corpo =
      [CmdSe
        (Tast.ExpOp ((MenorIgual, TipoBool),
          (Tast.ExpVar
            (VarSimples
              ("numero",
               {Lexing.pos_fname = ""; pos_lnum = 6; pos_bol = 88;
                pos_cnum = 96}),
             TipoInt),
           TipoInt),
          (Tast.ExpInt (0, TipoInt), TipoInt)),
        [CmdRetorno (Some (Tast.ExpInt (1, TipoInt)))],
        Some
         [CmdRetorno
           (Some
             (Tast.ExpOp ((Mult, TipoInt),
               (Tast.ExpVar
                 (VarSimples
                   ("numero",
                    {Lexing.pos_fname = ""; pos_lnum = 9; pos_bol = 139;
                     pos_cnum = 154}),
                  TipoInt),
```

```
                TipoInt),
               (Tast.ExpChamada ("fatorial",
                 [Tast.ExpOp ((Menos, TipoInt),
                   (Tast.ExpVar
                     (VarSimples
                       ("numero",
                        {Lexing.pos_fname = ""; pos_lnum = 9; pos_bol =
                           139;
                         pos_cnum = 172}),
                      TipoInt),
                     TipoInt),
                   (Tast.ExpInt (1, TipoInt), TipoInt))],
                 TipoInt),
                TipoInt))))])]}],
 [CmdSaida [Tast.ExpString ("Digite um numero", TipoString)];
  CmdEntrada
   [Tast.ExpVar
     (VarSimples
       ("numero",
        {Lexing.pos_fname = ""; pos_lnum = 14; pos_bol = 227;
         pos_cnum = 236}),
     TipoInt)];
  CmdAtrib
   (Tast.ExpVar
     (VarSimples
       ("fat",
        {Lexing.pos_fname = ""; pos_lnum = 15; pos_bol = 245;
         pos_cnum = 245}),
     TipoInt),
    Tast.ExpChamada ("fatorial",
     [Tast.ExpVar
       (VarSimples
         ("numero",
          {Lexing.pos_fname = ""; pos_lnum = 15; pos_bol = 245;
           pos_cnum = 260}),
       TipoInt)],
     TipoInt));
  CmdSaida
   [Tast.ExpString ("O fatorial de: ", TipoString);
    Tast.ExpVar
     (VarSimples
       ("numero",
        {Lexing.pos_fname = ""; pos_lnum = 16; pos_bol = 269;
         pos_cnum = 299}),
     TipoInt);
    Tast.ExpString (" \195\169", TipoString);
    Tast.ExpVar
     (VarSimples
       ("fat",
        {Lexing.pos_fname = ""; pos_lnum = 16; pos_bol = 269;
         pos_cnum = 312}),
     TipoInt)]]),
<abstr>)
```

8 Código de Três Endereços

O código de três endereços é composto por uma seqüência de instruções envolvendo operações binárias ou unárias e uma atribuição. O nome "três endereços"está associado à especificação, em uma instrução, de no máximo três variáveis: duas para os operadores binários e uma para o resultado.
Assim, expressões envolvendo diversas operações são decompostas nesse código em uma série de instruções, eventualmente com a utilização de variáveis temporárias introduzidas na tradução. Dessa forma, obtém-se um código mais próximo da estrutura da linguagem assembly e, conseqüentemente, de mais fácil conversão para a linguagem-alvo.
Uma possível especificação de uma linguagem de três endereços envolve quatro tipos básicos de instruções: expressões com atribuição, desvios, invocação de rotinas e acesso indexado e indireto.

8.1 Arquivos do Código de Três Endereços

8.1.1 Cod3End.ml

```
open Printf

open Ast
open Tast
open Codigo

let conta_temp = ref 0
let conta_rotulos = ref (Hashtbl.create 5)

let zera_contadores () =
  begin
    conta_temp := 0;
    conta_rotulos := Hashtbl.create 5
  end

let novo_temp () =
   let numero = !conta_temp in
   let _ = incr conta_temp in
   Temp numero

let novo_rotulo prefixo =
  if Hashtbl.mem !conta_rotulos prefixo
  then
     let numero = Hashtbl.find !conta_rotulos prefixo in
     let _ = Hashtbl.replace !conta_rotulos prefixo (numero + 1) in
     Rotulo (prefixo ^ (string_of_int numero))
  else
     let _ = Hashtbl.add !conta_rotulos prefixo 1 in
     Rotulo (prefixo ^ "0")

(* Codigo para impressão *)

let endr_to_str = function
   | Nome s -> s
   | ConstInt n -> string_of_int n
   | ConstFloat n -> string_of_float n
   | ConstChar n -> String.make 1 n
   | ConstString n -> n
```

```
   | Temp n  -> "t" ^ string_of_int n

let tipo_to_str t =
    match t with
      TipoInt -> "inteiro"
    | TipoString -> "string"
    | TipoBool -> "bool"
    | TipoVoid -> "void"
    | TipoReal -> "double"
    | TipoChar -> "char"
    | TipoArranjo (t,i,f) -> "arranjo"
    | TipoRegistro cs -> "registro"


let op_to_str op =
  match op with
  | Mais  -> "+"
  | Menos -> "-"
  | Mult  -> "*"
  | Div   -> "/"
  | Menor -> "<"
  | MenorIgual -> ">="
  | Igual -> "="
  | MaiorIgual -> ">="
  | Difer -> "!="
  | Maior -> ">"
  | And     -> "&&"
  | Or    -> "||"
  | Concat -> "^^"
  | Mod -> "%"
  | Expoente -> "^"

let rec args_to_str ats =
   match ats with
   | [] -> ""
   | [(a,t)] ->
     let str = sprintf "(%s,%s)" (endr_to_str a) (tipo_to_str t) in
     str
   | (a,t) :: ats ->
     let str = sprintf "(%s,%s)" (endr_to_str a) (tipo_to_str t) in
     str ^ ", " ^ args_to_str ats

let rec escreve_cod3 c =
  match c with
  | AtribBin (x,y,op,z) ->
      sprintf "%s := %s %s %s\n" (endr_to_str x)
                                 (endr_to_str y) (op_to_str (fst op)) (
                                    endr_to_str z)
  | Copia (x,y) ->
      sprintf "%s := %s\n" (endr_to_str x) (endr_to_str y)
  | Goto l ->
      sprintf "goto %s\n" (escreve_cod3 l)
  | If (x,l) ->
      sprintf "if %s goto %s\n" (endr_to_str x) (escreve_cod3 l)
  | IfFalse (x,l) ->
      sprintf "ifFalse %s goto %s\n" (endr_to_str x) (escreve_cod3 l)
  | IfRelgoto (x,oprel,y,l) ->
      sprintf "if %s %s %s goto %s\n" (endr_to_str x) (op_to_str (fst
         oprel))
```

```
                                        (endr_to_str y) (escreve_cod3 l)
  | Call (p,ats,t) -> sprintf "call %s(%s): %s\n" p (args_to_str ats) (
     tipo_to_str t)
  | Recebe (x,t) -> sprintf "recebe %s,%s\n" x (tipo_to_str t)
  | Local (x,t)  -> sprintf "local %s,%s\n" x (tipo_to_str t)
  | Global (x,t) -> sprintf "global %s,%s\n" x (tipo_to_str t)
  | CallFn (x,p,ats,t) ->
      sprintf "%s := call %s(%s): %s\n" (endr_to_str x) p (args_to_str ats
         ) (tipo_to_str t)
  | Return x ->
    (match x with
       None   -> "return\n"
     | Some x -> sprintf "return %s\n" (endr_to_str x) )
  | BeginFun (id,np,nl) -> sprintf "beginFun %s(%d,%d)\n" id np nl
  | EndFun -> "endFun\n\n"
  | Rotulo r -> sprintf "%s: " r


let rec escreve_codigo cod =
  match cod with
  | [] -> printf "\n"
  | c::cs -> printf "%s" (escreve_cod3 c);
             escreve_codigo cs

(* Código do tradutor para código de 3 endereço *)

let pega_tipo exp =
  match exp with
  | ExpVar (v, t) -> t
  | ExpInt (n, t) -> t
  | ExpReal (n, t) -> t
  | ExpString (n, t) -> t
  | ExpChar (n, t) -> t
  | ExpBool (n, t) -> t
  | ExpOp ((op,t),_,_) -> t
  | ExpChamada (id, args, t) -> t
  | _ -> failwith "pega_tipo: não implementado"


let rec traduz_exp exp =
  match exp with
  | ExpInt (n, TipoInt) ->
     let t = novo_temp () in
    (t, [Copia (t, ConstInt n)])

  | ExpReal (n, TipoReal) ->
     let t = novo_temp () in
    (t, [Copia (t, ConstFloat n)])

  | ExpChar (n, TipoChar) ->
     let t = novo_temp () in
    (t, [Copia (t, ConstChar n)])

  | ExpString (n, TipoString) ->
     let t = novo_temp () in
    (t, [Copia (t, ConstString n)])

  | ExpVar (v, tipo) ->
    (match v with
```

```
        VarSimples nome ->
        let id = fst nome in
        ((Nome id), [])
    )

  | ExpOp (op, exp1, exp2) ->
    let (endr1, codigo1) = let (e1,t1) = exp1 in traduz_exp e1
    and (endr2, codigo2) = let (e2,t2) = exp2 in traduz_exp e2
    and t = novo_temp () in
    let codigo = codigo1 @ codigo2 @ [AtribBin (t, endr1, op, endr2)] in
    (t, codigo)

  | ExpChamada (id, args, tipo_fn) ->
      let (enderecos, codigos) = List.split (List.map traduz_exp args) in
      let tipos = List.map pega_tipo args in
      let endr_tipos = List.combine enderecos tipos
      and t = novo_temp () in
      let codigo = (List.concat codigos) @
                   [CallFn (t, id, endr_tipos, tipo_fn)]
      in
        (t, codigo)
  | _ -> failwith "traduz_exp: não implementado"


let rec traduz_cmd cmd =
  match cmd with
  | CmdRetorno exp ->
    (match exp with
     | None -> [Return None]
     | Some e ->
       let (endr_exp, codigo_exp) = traduz_exp e in
       codigo_exp @ [Return (Some endr_exp)]
    )
  | CmdAtrib (elem, ExpInt (n, TipoInt)) ->
    let (endr_elem, codigo_elem) = traduz_exp elem
    in codigo_elem @ [Copia (endr_elem, ConstInt n)]

  | CmdAtrib (elem, exp) ->
    let (endr_exp, codigo_exp) = traduz_exp exp
    and (endr_elem, codigo_elem) = traduz_exp elem in
    let codigo = codigo_exp @ codigo_elem @ [Copia (endr_elem, endr_exp)]
    in codigo

  | CmdSe (teste, entao, senao) ->
    let (endr_teste, codigo_teste) = traduz_exp teste
    and codigo_entao = traduz_cmds entao
    and rotulo_falso = novo_rotulo "L" in
    (match senao with
        | None -> codigo_teste @
                  [IfFalse (endr_teste, rotulo_falso)] @
                  codigo_entao @
                  [rotulo_falso]
        | Some cmds ->
          let codigo_senao = traduz_cmds cmds
          and rotulo_fim = novo_rotulo "L" in
              codigo_teste @
              [IfFalse (endr_teste, rotulo_falso)] @
              codigo_entao @
              [Goto rotulo_fim] @
```

```
            [rotulo_falso] @ codigo_senao @
            [rotulo_fim]
  )
| ComandoExpress (ExpChamada (id, args, tipo_fn)) ->
    let (enderecos, codigos) = List.split (List.map traduz_exp args) in
    let tipos = List.map pega_tipo args in
    let endr_tipos = List.combine enderecos tipos in
    (List.concat codigos) @
    [Call (id, endr_tipos, tipo_fn)]

| ComandoExpress _ -> []

| CmdSaida args ->
    let (end1,cod) = (traduz_exp (ExpString ("\\n", TipoString))) in
    let (enderecos, codigos) = List.split (List.map traduz_exp args) in
    let tipos = List.map pega_tipo args in
    let endr_tipos = (List.combine enderecos tipos)@[(end1,TipoString)]
       in
    cod @
    (List.concat codigos) @
    [Call ("print", endr_tipos, TipoVoid)]

| CmdEntrada arg ->
    let (endereco, codigo) =  (traduz_exp arg) in
    let tipo = pega_tipo arg in
    let endr_tipo = List.combine [endereco] [tipo] in
    (List.concat [codigo]) @
    [Call ("read", endr_tipo, TipoVoid)]

(* | CmdEntrada args ->
    let (end1,cod) = (traduz_exp (ExpString ("\\n", TipoString))) in
    let (enderecos, codigos) = List.split (List.map traduz_exp args) in
    let tipos = List.map pega_tipo args in
    let endr_tipos = List.combine enderecos tipos in
    cod @
    (List.concat codigos) @
    [Call ("read", endr_tipos, TipoVoid)] @
    [Call ("print", [(end1,TipoString)], TipoVoid)] *)


| CmdWhile (teste, doit) ->
  let (endr_teste, codigo_teste) = traduz_exp teste
  and codigo_doit = traduz_cmds doit
  and rotulo_inicio = novo_rotulo "W"
  and rotulo_fim = novo_rotulo "W"  in
    [rotulo_inicio] @ codigo_teste         @ [IfFalse (endr_teste,
       rotulo_fim)] @
    codigo_doit     @ [Goto rotulo_inicio] @ [rotulo_fim]

| CmdFor (variavel, exp, condicao, var,inc, doit) ->
  let (endr_teste, codigo_teste) = traduz_exp condicao
  and codigo_atrib = traduz_cmds [CmdAtrib (variavel, exp)]
  and codigo_doit = traduz_cmds ( List.append doit [CmdAtrib (var, inc)]
      )
  and rotulo_inicio = novo_rotulo "W"
  and rotulo_fim = novo_rotulo "W" in
    codigo_atrib @ [rotulo_inicio]      @ codigo_teste @ [IfFalse (
       endr_teste, rotulo_fim)] @
    codigo_doit  @ [Goto rotulo_inicio] @ [rotulo_fim]
```

```
  | CmdCase (teste, cases, default) ->
  let rotulo_fim = novo_rotulo "L" in
  (* gera_cases traduz cada comando da lista de cases um por vez
    e traduz os cases da cauda recursivamente.
    O caso base, (quando a lista está vazia) acontece quando toda a lista
    já foi percorrida e deve-se gerar a tradução para a parte "default" do
        comando switch*)
    let rec gera_cases cases = (match cases with
        Case (l, c)::tail ->let rotulo_proxima = novo_rotulo "C" in
          let codigo_doit = traduz_cmds c in
          let (end_exp, codigo_exp) = traduz_exp
          (* Aqui é gerado o teste de igualdade para cada case.
              Note que é necessário otimizar esse código, pois ele
                 interpreta "teste"
              a cada Case desnecessariamente *)
          (* pega_tipo teste é necessário para montar a expressão, note
             que não estamos preocupados
            em qual tipo é esse, pois no módulo semântico, já foi
               verificado que este é válido e
            é o mesmo para operador e operandos *)
          (ExpOp ((Igual, (pega_tipo teste)), (teste, (pega_tipo teste)),
             (l, (pega_tipo teste)))) in
          codigo_exp
          @ [IfFalse (end_exp, rotulo_proxima)]
          @ codigo_doit
           (* Nesta implementação, pode haver dois casses com o mesmo
              valor, portanto ambos devem ser executados.
            Caso queira mudar para executar somente o primeiro que casar
               com a expressão de teste,
            descomente a próxima linha *)
          (* @ [Goto rotulo_fim]   *)
          @ [rotulo_proxima]
          @ (gera_cases tail)
        | [] -> (match default with
            None -> []
            | Some cmds_default -> traduz_cmds cmds_default
          (* Aqui não é necessário um Goto rotulo_fim,
            pois tal rótulo já se encontra logo em seguida no código *)
        )
    )
     in (gera_cases cases) @ [rotulo_fim]



and traduz_cmds cmds =
  match cmds with
  | [] -> []
  | cmd :: cmds ->
     let codigo = traduz_cmd cmd in
     codigo @ traduz_cmds cmds

let traduz_fun ast =
  let trad_local x =
    match x with
      DecVar ((id,pos),t) -> Local (id,t)
  in
  match ast with
```

```
    Funcao {fn_nome; fn_tiporet; fn_formais; fn_locais; fn_corpo} ->
    let nome = fst fn_nome
    and formais = List.map (fun ((id,pos),tipo) -> Recebe (id,tipo))
        fn_formais
    and nformais = List.length fn_formais
    and locais = List.map trad_local fn_locais
    and nlocais = List.length fn_locais
    and corpo = traduz_cmds fn_corpo
    in
    [BeginFun (nome,nformais,nlocais)] @ formais @ locais @ corpo @ [
        EndFun]


let tradutor ast_tipada =
  let trad_global x =
    match x with
      DecVar ((id,pos),t) -> Global (id,t)
  in
  let _ = zera_contadores () in
  let (Programa (decs_globais, decs_funs, corpo)) = ast_tipada in
  let globais_trad = List.map trad_global decs_globais in
  let funs_trad = List.map traduz_fun decs_funs in
  let corpo_trad = traduz_cmds corpo in
  globais_trad @ (List.concat funs_trad) @
  [BeginFun ("main",0,0)] @ corpo_trad @ [EndFun]
```

8.1.2 Cod3EndTest.ml

```
open Printf
open Lexing

open Ast
exception Erro_Sintatico of string

module S = MenhirLib.General (* Streams *)
module I = Sintatico.MenhirInterpreter

open Semantico
open Codigo
open Cod3End

let message =
  fun s ->
    match s with
    | 0 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 1 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 34 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 35 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 36 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 72 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 47 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
```

```
    | 48 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 49 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 51 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 52 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 55 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 56 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 57 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 58 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 61 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 62 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 63 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 64 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 73 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 74 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 95 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 89 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 97 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 98 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 99 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 65 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 66 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 53 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 67 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 68 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 59 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 60 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 42 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 41 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 70 ->
        "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
    | 75 ->
```

```
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 77 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 76 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 105 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 84 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 43 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 85 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 86 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 45 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 46 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 102 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 103 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 81 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 3 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 2 ->
    "<YOUR SYNTAX ERROR MESSAGE HERE>\n"
| 6 ->
    "estado 6: esperava um tipo. Exemplo:\n   x : inteiro;\n"
| 7 ->
    "estado 7: esperava a definicao de um campo. Exemplo:\n   i:
       registro\n         parte_real: inteiro;\n         parte_imag:
       inteiro;\n      fim registro;\n      "
| 8 ->
    "estado 8: esperava ':'. Exemplo:\n   x: inteiro;\n   "
| 9 ->
    "estado 9: esperava um tipo. Exemplo:\n   x: inteiro;\n"
| 25 ->
    "estado 25: esperva um ';'.\n"
| 26 ->
    "estado 26: uma declaracao foi encontrada. Para continuar era\n
       esperado uma outra declara\195\167\195\163o ou a palavra '
       inicio'.\n"
| 29 ->
    "estado 29: espera a palavra 'registro'. Exemplo:\n   i: registro\
       n         parte_real: inteiro;\n         parte_imag: inteiro;\n
             fim registro;\n"
| 31 ->
    "estado 31: esperava um ';'. \n"
| 107 ->
    "estado 107: uma declaracao foi encontrada. Para continuar era\n
         esperado uma outra declara\195\167\195\163o ou a palavra '
       inicio'.\n"
| 13 ->
    "estado 13: esperava um '['. Exemplo:\n   arranjo [1..10] de
       inteiro;\n"
| 14 ->
```

```
        "estado 14: esperava os limites do vetor. Exemplo:\n   arranjo
           [1..10] de inteiro;\n"
    | 15 ->
        "estado 15: esperava '..'. Exemplo:\n   1 .. 10\n"
    | 16 ->
        "estado 16: esperava um numero inteiro. Exemplo:\n   1 .. 10\n"
    | 18 ->
        "estado 18: esperava um ']'. Exemplo\n   arranjo [1..10] de
           inteiro;\n"
    | 19 ->
        "estado 19: esperava a palavra reservada 'de'. Exemplo:\n
           arranjo [1..10] de inteiro;\n"
    | 20 ->
        "estado 20: esperava um tipo. Exemplo\n   arranjo [1..10] de
           inteiro;\n"
    | _ ->
        raise Not_found

let posicao lexbuf =
    let pos = lexbuf.lex_curr_p in
    let lin = pos.pos_lnum
    and col = pos.pos_cnum - pos.pos_bol - 1 in
    sprintf "linha %d, coluna %d" lin col

(* [pilha checkpoint] extrai a pilha do autômato LR(1) contida em
   checkpoint *)

let pilha checkpoint =
  match checkpoint with
  | I.HandlingError amb -> I.stack amb
  | _ -> assert false (* Isso não pode acontecer *)

let estado checkpoint : int =
  match Lazy.force (pilha checkpoint) with
  | S.Nil -> (* O parser está no estado inicial *)
     0
  | S.Cons (I.Element (s, _, _, _), _) ->
     I.number s

let sucesso v = Some v

let falha lexbuf (checkpoint : (Sast.expressao Ast.programa) I.checkpoint)
    =
  let estado_atual = estado checkpoint in
  let msg = message estado_atual in
  raise (Erro_Sintatico (Printf.sprintf "%d - %s.\n"
                                        (Lexing.lexeme_start lexbuf) msg))

let loop lexbuf resultado =
  let fornecedor = I.lexer_lexbuf_to_supplier Lexico.token lexbuf in
  I.loop_handle sucesso (falha lexbuf) fornecedor resultado


let parse_com_erro lexbuf =
  try
    Some (loop lexbuf (Sintatico.Incremental.programa lexbuf.lex_curr_p))
  with
  | Lexico.Erro msg ->
     printf "Erro lexico na %s:\n\t%s\n" (posicao lexbuf) msg;
```

```
      None
  | Erro_Sintatico msg ->
     printf "Erro sintático na %s %s\n" (posicao lexbuf) msg;
     None

let parse s =
  let lexbuf = Lexing.from_string s in
  let ast = parse_com_erro lexbuf in
  ast

let parse_arq nome =
  let ic = open_in nome in
  let lexbuf = Lexing.from_channel ic in
  let ast = parse_com_erro lexbuf in
  let _ = close_in ic in
  ast

let verifica_tipos nome =
  let ast = parse_arq nome in
  match ast with
    Some (Some ast) -> semantico ast
  | _ -> failwith "Nada a fazer!\n"

let traduz nome =
  let (arv,tab) = verifica_tipos nome in
  tradutor arv

let imprime_traducao cod =
   let _ = printf "\n" in
   escreve_codigo cod
```

8.2 Para compilar:

.ocamlinit

```
let () =
  try Topdirs.dir_directory (Sys.getenv "OCAML_TOPLEVEL_PATH")
  with Not_found -> ()
;;

#use "topfind";;
#require "menhirLib";;
#directory "_build";;
#load "sintatico.cmo";;
#load "lexico.cmo";;
#load "ast.cmo";;
#load "sast.cmo";;
#load "tast.cmo";;
#load "tabsimb.cmo";;
#load "ambiente.cmo";;
#load "semantico.cmo";;
#load "Codigo.cmo";;
#load "Cod3End.cmo";;
#load "cod3endTest.cmo";;

open Ast
open Ambiente
open Codigo
```

```
open Cod3End
open Cod3endTest
```

No Terminal

```
    ocamlbuild -use-ocamlfind -use-menhir -menhir "menhir --table" -
       package menhirLib cod3endTest.byte

##Para usar, entre no ocaml

    rlwrap ocaml


##e se desejar ver apenas a árvore sintática que sai do analisador sintá
   tico, digite

     parse_arq "exemplos/Tipos/ex8.tip";;

## Depois, para ver a saída do analisador semântico já com a árvore
   anotada com

   o tipos, digite:

   verifica_tipos "exemplos/Tipos/ex8.tip";;

 ##Note que o analisador semântico está retornando também o ambiente
    global. Se quiser separá-los, digite:

   let (arv, amb) = verifica_tipos "exemplos/Tipos/ex8.tip";;

## Para ver o código de 3 endereços:

   traduz "exemplos/Tipos/ex8.tip";;

## ou se quiser ver em um formato mais legível:

    let cod = traduz "exemplos/Tipos/ex8.tip" in imprime_traducao cod;;
```

8.2.1 Testes

Teste micro10

```
# let cod = traduz "micro10.js" in imprime_traducao cod;;

global numero,inteiro
global fat,inteiro
beginFun fatorial(1,0)
recebe numero,inteiro
t0 := 0
t1 := numero >= t0
ifFalse t1 goto L0:
t2 := 1
return t2
goto L1:
L0: t3 := 1
t4 := numero - t3
t5 := call fatorial((t4,inteiro)): inteiro
t6 := numero * t5
return t6
```

```
L1: endFun

beginFun main(0,0)
t7 := \n
t8 := Digite um numero
call print((t8,string), (t7,string)): void
call read((numero,inteiro)): void
t9 := call fatorial((numero,inteiro)): inteiro
fat := t9
t10 := \n
t11 := O fatorial de:
t12 :=  é
call print((t11,string), (numero,inteiro), (t12,string), (fat,inteiro), (
    t10,string)): void
endFun


- : unit = ()
```

Teste micro08

```
# let cod = traduz "micro08.js" in imprime_traducao cod;;

beginFun main(0,1)
local numero,inteiro
numero := 1
W0: t0 := 0
t1 := numero != t0
ifFalse t1 goto W1:
t2 := \n
t3 := Digite um numero:
call print((t3,string), (t2,string)): void
call read((numero,inteiro)): void
t4 := 10
t5 := numero > t4
ifFalse t5 goto L0:
t6 := \n
t7 := O numero
t8 := eh maior que 10
call print((t7,string), (numero,inteiro), (t8,string), (t6,string)): void
goto L1:
L0: t9 := \n
t10 := O numero
t11 := eh menor que 10
call print((t10,string), (numero,inteiro), (t11,string), (t9,string)):
    void
L1: goto W0:
W1: endFun

beginFun main(0,0)
call main(): void
endFun


- : unit = ()
```

Referências

1. Node: https://nodejs.org/en/
2. EMSDK: https://webassembly.org/getting-started/developers-guide/
3. WebAssembly: https://github.com/WebAssembly/wabt
4. WebAssembly: https://mbebenita.github.io/WasmExplorer/
5. Monografia: Monografia referência PedroJavascriptWebassembly
6. Aula: aulaLexico.pdf
7. Real World OCaml: https://v1.realworldocaml.org/v1/en/html/index.html
8. JavaScript: https://www.devmedia.com.br/guia/javascript/34372
9. Aula: aulaSintatico.pdf
10. Aula: aulaMenhir.pdf
11. Aula: aulaSemantico.pdf
12. Aula: Interprete
13. Aula: Representação Intermediaria e Código de Três Endereços